

NASA Complex Electronics Guidebook for Assurance Professionals

Author:
Kalynda Berens
for NASA Code Q

Date:
December 31, 2004

Table of Contents

1	Introduction	1
1.1	Purpose of this guidebook.....	1
1.2	Scope of the guidebook.....	1
1.3	Anticipated Audience.....	2
1.4	Layout of guidebook	2
2	Assurance Issues with Complex Electronics	3
2.1	Blurring the hardware/software line	3
2.2	Concerns and issues	5
3	Complex Electronics Overview	7
4	Design Process	18
4.1	Overview of the Complex Electronics Design Process.....	18
4.2	Requirements and Specifications	22
4.3	Design Entry.....	23
4.3.1	Abstraction and Modeling	26
4.4	Hardware Description Languages	28
4.4.1	Overview of Hardware Description Languages	28
4.4.2	Programming Example	35
4.5	Synthesis.....	38
4.6	Implementation.....	41
4.7	Verification	47
5	Process Assurance	52
5.1	Process Assurance Overview.....	52
5.1.1	Why do Process Assurance?	52
5.1.2	Process Assurance for Complex Electronics.....	53
5.1.3	Tools of the Process Assurance Trade	54
5.2	Identifying Complex Electronics	56
5.3	Process Assurance activities.....	58
5.3.1	Project Conception	59
5.3.2	Requirements	61
5.3.3	Design Entry.....	61
5.3.4	Design Synthesis.....	63
5.3.5	Implementation.....	63
5.3.6	Testing	64
5.3.7	Operations and Maintenance	65
5.3.8	Supporting Processes	65
6	Future Trends.....	67
6.1	Changes in CE design and verification.....	67
6.2	NASA assurance changes	69
	Appendix A.....	71
A.1	Acronyms	71
A.2	Glossary	72
A.3	Links	78
	Appendix B.....	80
B.1	CPLD.....	80

NASA Complex Electronics Guidebook for Assurance Professionals

B.2	FPGA	85
B.3	ASIC.....	91
B.4	SoC	97
B.5	Reconfigurable Computing.....	103

1 Introduction

1.1 Purpose of this guidebook

Complex electronics (CE) encompasses programmable and designable complex integrated circuits. “Programmable” logic devices can be programmed by the user and range from simple chips to complex devices capable of being programmed on-the-fly. Some types of programmable devices this guidebook will address are:

- Field Programmable Gate Arrays (FPGA)
- Complex Programmable Logic Device (CPLD)
- System-on-chip (SoC)

“Designable” logic devices are integrated circuits that can be designed but not programmed by the user. The design is submitted to a manufacturer for implementation in the device. Application-Specific Integrated Circuit (ASIC) is an example of a designable device.

Assurance activities for complex electronics are often lagging behind the pace of the technology. These devices are commonly used within NASA systems, sometimes in safety-critical systems. Both software and quality assurance engineers need to understand what these devices are, where they are used, and how they are designed. Detailed assurance guidance is not available, as the NASA Office of Safety and Mission Assurance is only starting to address these devices. This guidebook provides some general suggestions that, if applied, may increase confidence in the quality of complex electronic devices.

1.2 Scope of the guidebook

This guidebook will provide an overview of complex electronics, the design process, and assurance activities. It is meant as an introductory text that will inform the reader.

You will hopefully learn:

- Which devices are “complex electronics”, and which are not
- Fascinating information about each of the devices, including NASA projects that are using the devices
- How electronics engineers design and program the devices
- Assurance and verification activities for complex electronics
- Where NASA is going with assurance activities for complex electronics

Additional assurance activities for complex electronics devices may be required in the future. This course *will not* prepare you to perform those activities. It *will* provide you with a general understanding of the devices and the design and assurance activities. You will be able to “speak the language” when communicating with the hardware design engineers.

1.3 Anticipated Audience

This guidebook is primarily intended for software and quality assurance engineers who do not have significant experience with complex electronics. You don't need a hardware background to understand the material in this guidebook. However, being familiar with embedded systems or flight hardware may help you absorb some of the concepts.

System safety personnel are encouraged to review this guidebook. Modern technology, especially electronics, is changing at a rapid pace. Projects and systems you support will be using these devices in the near future, if they are not already doing so.

1.4 Layout of guidebook

Section 1 provides the purpose, scope, and layout for the guidebook.

Section 2 describes why assurance engineers need to be aware of complex electronics and details some concerns and issues with the current state of assurance activities.

Section 3 gives an overview of complex electronics.

Section 4 describes the design process for complex electronics. A short description of hardware description languages, along with a simple example, are included.

Section 5 provides an overview of current and suggested assurance practices for complex electronics. This section also contains an overview of process assurance.

Section 6 describes some new areas in design and verification of complex electronics, as well as NASA policy and possible changes to that policy.

Appendix A provides an acronym list, a glossary, and links for further information.

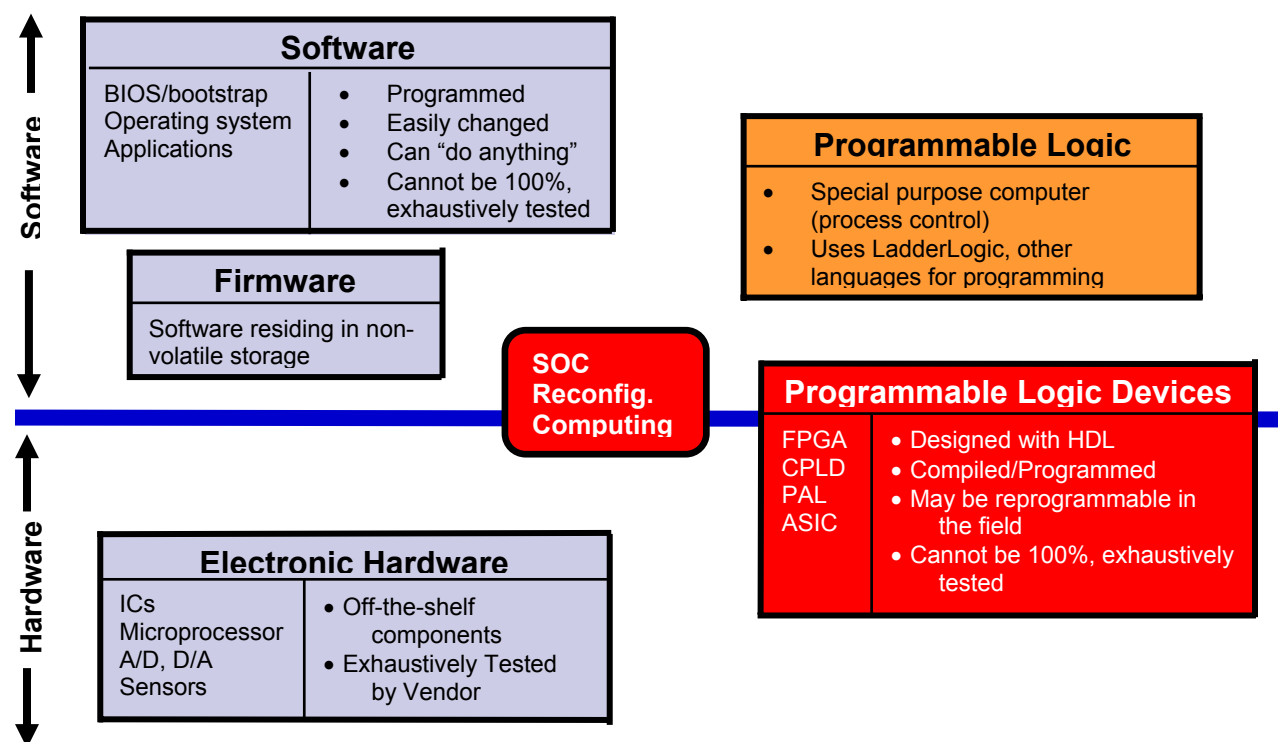
Appendix B describes each of the types of complex electronics in detail.

2 Assurance Issues with Complex Electronics

2.1 Blurring the hardware/software line

Programmable Logic devices are now blurring the hardware/software boundary. Field Programmable Gate Arrays (FPGAs) can have from 30,000 to over a million logic gates. Systems on Chip (SoC) devices combine a microprocessor, input and output channels, and often an FPGA for programmability. These devices can now be programmed to perform tasks that were previously handled in software, such as communication protocols. With increased complexity, the possibility of “software-like” bugs (incorrect logic) and unexpected interactions is more likely. It is vital to be able to assure that the systems are designed and implemented correctly, tested fully, and are reliable.

The figure below shows the relationship of software, firmware, Programmable Logic Controllers (PLCs), electronics hardware, and complex electronics (the items in the red boxes). Boxes above the boundary line are software and those below the line are hardware. Complex electronics straddles the line to various degrees.



The Federal Aviation Administration (FAA) has become concerned about the usage of complex electronic hardware in aviation. A study in 1995 stated, “There are no techniques and methods of

design, documentation, testing, and verification identified or recognized by the Federal Aviation Administration (FAA) for today's complex hardware designs." Since that time, the FAA has worked with other organizations to develop DO-254, "Design Assurance Guidance for Airborne Electronic Hardware", which provides guidelines on the use of process assurance for complex electronic hardware. One motivating factor for DO-254 is that organizations were implementing software functions in FPGAs and ASICs to avoid the need to follow the software assurance/safety standard, DO-178B!

The pace of technological change and the new uses that people find for current technology are strong motivators for NASA to begin to define acceptable assurance practices for complex electronics. An example of an assurance nightmare is adaptive or reconfigurable computing, which is computers, chips, or systems that alter their functionality to adapt to changing applications. Adaptive computing is usually implemented with FPGAs and allows for parallel processing. Adaptive computing is expected to be the next "breakthrough" in computing. Many applications of the technique for the military are being proposed, and adaptive computing is likely to be used in space systems.

How does Programmable Logic differ from Firmware?

Firmware has various definitions, but the most common is that found in IEEE 610.12-1990: "The combination of hardware device and computer instructions and data that reside as read-only software on that device." In other words, it is software that is placed in a read-only device, such as an EPROM, from which it may be read or copied. The EPROM acts solely as a storage device, much like a disk. Thus, firmware is simply stored software.

Complex electronics, such as FPGAs and ASICs, are not firmware (as defined by IEEE 610.12), because what resides in them is not a software program. Instead, software is used to define the logic structure for a hardware device, which is what these devices become once they are programmed. These devices are better thought of as hybrid hardware/software devices, or "soft hardware".

Some types of complex electronics are even harder to define, such as:

- System-on-Chip (SoC) is an electronic chip (FPGA or ASIC) that contains gates and logic elements. These elements usually include a microprocessor core, plus peripheral devices such as analog-to-digital converters. SoCs may include embedded software (i.e. firmware) as part of the device.
- FPGAs are "soft hardware", except when they are used in reconfigurable or adaptable computing. In that case, they are part of a complex system that is reprogrammed on the fly. The FPGAs replace a microprocessor, and the act of reprogramming them (and the logic that determines the activities) is the "software" of the system.

Programmable logic devices blur the hardware/software boundary. The IEEE definition of firmware does not encompass these devices. Either an updated definition of firmware needs to be adopted by NASA, or a new term needs to be coined to describe these devices.

Comparing Complex Electronics and Software

Complex electronics devices do not work in the same way as software. The main difference is that software is serial (one activity is performed after another) and hardware is parallel (multiple operations occur at the same time). It is very important to always remember that the ultimate result of a programmable logic device is hardware. Hardware “programming” languages, such as VHDL, can be thought of as a virtual or abstract piece of hardware.

However, similarities exist between programming languages for complex electronics (e.g. Verilog or VHDL) and software languages. VHDL, for example, is based on Ada syntax, has data types common to most higher-level languages, uses objects (e.g. constants and variables), and has sequential statements.

A software assurance engineer reviewing programmable logic “code” should not be lulled by the similarities to “regular” programming languages. Complex electronics and programmable logic devices are ultimately hardware, and those differences must be acknowledged. Additional training (beyond this course) would be required before software assurance personnel could provide quality assurance for these devices.

2.2 Concerns and issues

Verification Issues with Complex Electronics

Verification means that you’ve demonstrated that the system or sub-system meets the requirements you’ve specified. Complex systems, especially those including software, are notoriously hard to adequately verify. Complex electronics adds additional verification concerns to the mix:

- Tool-induced design errors occur and can be difficult to detect. Tools are a vital part of complex electronics design, and the designer often does not know what errors a tool could potentially produce.
- Complex functionality cannot be completely simulated, nor the resulting chip completely tested.
- It can be difficult to detect faulty operation of complex electronics due to design or tool-induced errors, unexpected interactions, or even defects in the silicon.
- Due to extremely small ASIC geometries, certain analog and transmission line phenomena occur internal to the ASIC, generating failures that are data-sensitive. Designers and tools may not account for these effects, which can easily escape notice during test.

Assurance Issues with Complex Electronics

Besides problems with testing and verifying the designs and implementations of complex electronics, quality assurance is struggling with how to adequately deal with the “software-like” aspects of these devices. Some problems and concerns are:

- ASICs and FPGAs have been used to avoid the rigors of the software approval process. This results in fundamental verification matters being bypassed
- Complex Electronic devices are designed and programmed by engineers, often without quality assurance oversight or configuration management control of the designs. In addition, the development process may not be well defined or followed.
- ASICs, FPGAs, and System-on-Chip (SoC) can contain embedded microprocessor cores with user-supplied software. They combine electronics and firmware into one chip. The presence of this firmware (i.e. software) is not always obvious to assurance personnel.
- High-level languages (e.g. C, C++) are now being used to define complex electronic designs (in whole or in part).
- Hardware QA may not be fully cognizant of the functions, potential problems, and issues with these devices.
- Software Assurance personnel are currently not trained to understand complex electronics, and may not be able to provide effective oversight and assurance.
- Meaningful verification efforts require the person performing the verification to be knowledgeable about the complex electronic device and the tool suite used to create and implement the design.

3 Complex Electronics Overview

“Complex electronics” is a term applied to various forms of programmable or designable hardware devices. The two elements of the term - complex and electronics - can be used to help distinguish what devices are, or are not, of interest.

Programmable versus Designable Devices

Programmable Logic Devices (PLDs) are hardware integrated circuits that are programmable by the user. They contain configurable logic and flip-flops, which are linked together with programmable interconnects. Memory cells control and define the function that the logic performs and how the various logic functions are interconnected. PLDs can be divided into various categories, and range from simple devices to complex devices capable of being programmed on-the-fly. Devices in this category include

- Programmable Array Logic (PAL)
- Generic Array Logic (GAL)
- Programmable Logic Array (PLA)
- Complex Programmable Logic Device (CPLD)
- Field Programmable Gate Array (FPGA)

Some integrated circuits can be designed by the user, and submitted to a manufacturer for creation of multiple copies. This allows specialty circuits to be designed for a device, such as a cell phone. Once created, the devices cannot be reprogrammed by the user. ASICs and System-on-Chip (SoC) are examples of designable devices. We’ll get into more on each of these devices later, or you can follow the hyperlinks for immediate details.

Complex Electronics. The *electronics* part of this term is fairly easy to identify - if you need electrons to make it run (aka power), it’s electronics. So all the devices listed above qualify. So do off-the-shelf integrated circuits (ICs), microprocessors, logic gates, analog-to-digital converters, buffers and other components. Do we care about all these devices? The quick answer is “no”. We only care about *complex* electronics.

The *complex* adjective is used to distinguish between simple devices, such as off-the-shelf ICs and logic gates, and user-creatable devices. How do we determine what devices fall into *simple* or *complex*? We’ll get into that later. For now, a good rule of thumb is, if you can program or design the internal logic of the device and it has more than a few gates and connections, it is probably *complex*.

Does firmware fall under this category? Firmware has various definitions, but the most common is that found in IEEE 610.12-1990: “The combination of hardware device and computer instructions and data that reside as read-only software on that device.” In other words, it is software that is placed in a read-only device, such as an EPROM or Flash, from which it may be read or copied. The EPROM acts solely as a storage device, much like a disk. The software may

NASA Complex Electronics Guidebook for Assurance Professionals

be complex and reside on electronic components, but it does not affect the internal logic or configuration of the chips. Firmware is not considered complex electronics.

So, before we get into more details on complex electronics, let's give some examples:

Device	User Interactions	Complex Electronics?	Why or Why Not?
Complex Programmable Logic Device	Define and program the internal logic elements	Yes	Electronic and complex
FIFO	Use it	No	Electronic, but not complex
Microprocessor	Execute software instructions on it to perform arithmetic and other operations.	No	The software executes pre-defined commands. It does not change the internal logic arrangement of the microprocessor.
Software	Design, develop, execute	No	Definitely complex, but not electronic
Application Specific Integrated Circuit (ASIC)	Design, Use resulting chip	Yes (above a threshold)	Most ASICs are complex. It's possible to make a simple ASIC; but why bother? Such devices are likely to be already available.
EEPROM (Electrically Erasable Programmable Read-Only Memory)	Program the device with data or software	No	The device itself is not complex. The software or data does not change the internal logic of the device.

A bit of history

Now that your head is swimming from the acronyms and ideas, let's go back to the start and explain each device. You'll get a little history as well, to help put everything in context.

The story starts with the development of discrete logic. Each logic chip had a purpose (e.g AND gate, OR gate, flip-flop) and could be wired together with other chips to make the desired circuit. Pinouts on the chip were fixed. Manufacturing such a system took a lot of time because each design change required that the wiring be redone. This usually meant building a new printed circuit board.

The chip makers solved the problem of time-consuming rewiring for design changes by placing an unconnected array of AND-OR gates in a single chip called a *programmable logic device* (PLD). The PLD contained an array of fuses that could be blown open or left closed to connect various inputs to each AND gate. You could program a PLD to perform the logic functions you needed in your system. Since the PLDs could be rewired internally, there was less of a need to change the printed circuit boards which held them.

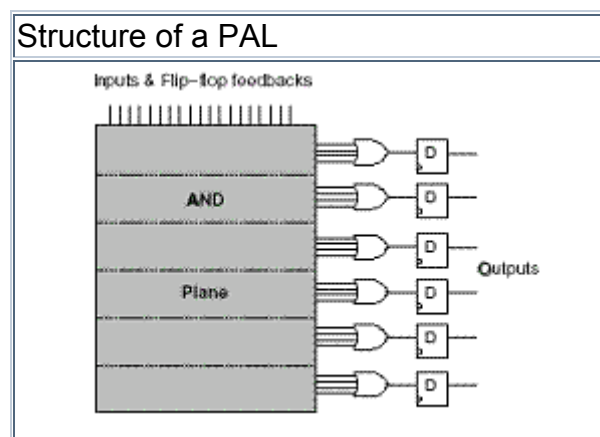
Simple Programmable Logic Devices

Simple PLDs come in a variety of flavors. They are called *simple* to distinguish them from the Complex PLDs (CPLDs, discussed below), and because they are actually pretty simple devices, as modern integrated circuits go.

Programmable Array Logic

Programmable Array Logic (PAL) chips are a family of *fuse-programmable* integrated circuits originally developed by MMI (Monolithic Memories, Inc.). *Logic* means that the chips allow the user to program a set of AND and OR gates (or NAND/NOR) to create the desired logic sequence. PALs consist of a programmable AND array followed by a non-programmable OR array. Inputs are fed into the AND array, which performs the desired AND functions and generates product terms, which are then fed into the OR array. In the OR array, the outputs of the various product terms are combined to produce the desired outputs.

Using a fixed number of OR gates, rather than a completely programmable set, allows the device to be fast. The high speed available in PALs makes them still popular today, despite the abundance of newer chips.



Fuse-programmable has to do with how PALs are programmed. Connections between the gates in a PAL are made using fuses that are either connected or disconnected (blown). Overvoltage (above the operational limits of the chip) is used to blow the fuses for the connections that are not desired. This operation is permanent, so once programmed, a PAL cannot be reprogrammed.

Fuse maps, which determine what fuses are, or are not, blown for a particular PAL can be generated in several ways. Languages such as PALASM or CUPL can be used, with the resulting

logic design complied into JEDEC ASCII/ hexadecimal files. Modern support software for PALs allows a direct translation from a schematic, truth table or state table to the fuse map. Some software even accepts timing diagrams as input. Hardware description languages (HDL) can also be used to synthesize the fuse map. However the map is created, it must be provided as input to a special electronic programming system, available from either the manufacturer or a third-party, for physical programming of the chip.

Generic Array Logic

Generic Array Logic (GAL) was introduced by Lattice Semiconductor. A GAL consists of a reprogrammable AND array, a fixed OR array, and reprogrammable output logic. Electrically Erasable Programmable Read-Only Memory (EEPROM) is used, rather than fuses, to provide the connections. This allows the GAL to be erased and reprogrammed.

The GAL is very useful in the prototyping stage of a design, when any bugs in the logic can be corrected by reprogramming. GALs are programmed and reprogrammed using a PAL programmer, and the same types of languages or processes used for PAL chips. If speed is important (and it usually is), a PAL can be used, once the design is finalized.

Programmable Logic Array

Programmable Logic Array (PLA) devices differ from PALs in the OR-gates area. PALs could only be programmed in the AND-plane. With PLA chips, a set of programmable AND planes are linked to a set of programmable OR planes, which can then be conditionally complemented to produce an output. PLA devices allow far more design options than PALs, but the downside is reduced performance.

Like PALs, PLA devices are fuse-based and can be programmed only once. Tools and languages are readily available to translate a logic design into the fuse map required for PLA programming.

Here's a comparison of the simple programmable devices:

	PROM	PAL	GAL	PLA
Input lines	hard-wired	programmable	programmable	programmable
Output lines	programmable	hard-wired	hard-wired	programmable
Versatility	low	moderate	moderate	high
Difficulty in manufacturing, programming and testing	low	moderate	low	high
Reprogrammable?	No	No	Yes	No

Complex Programmable Logic Devices (CPLD)

Simple PLDs can only handle up to 10 to 20 logic equations, which is not a very large logic design. Designers need to figure out how to break a larger design apart and fit the pieces into a

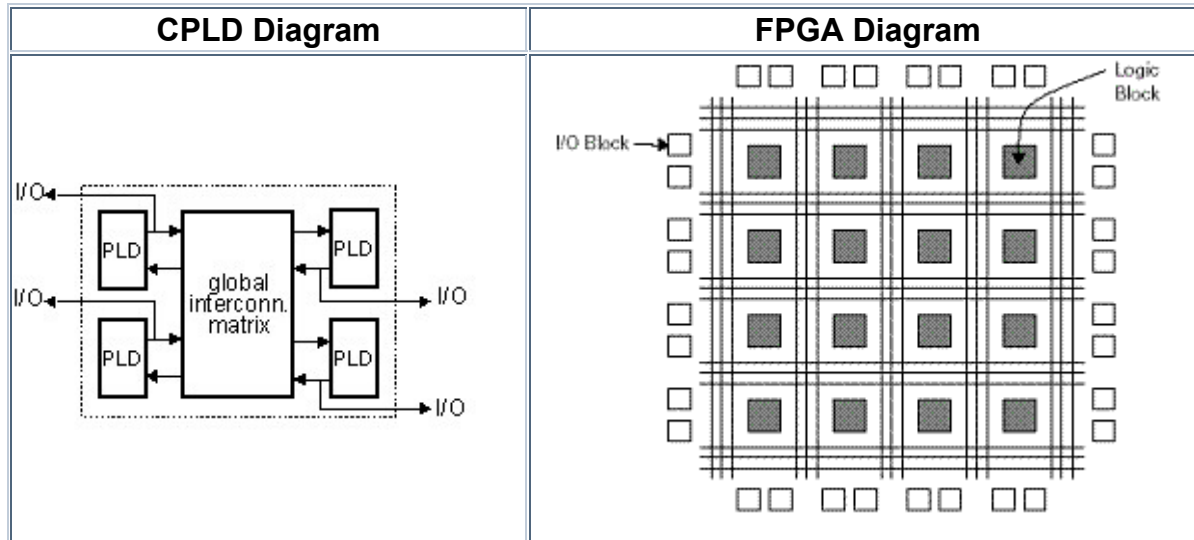
set of PLDs. This is a time-consuming process, and means you have to interconnect the PLDs with wires. Like the old days, when there was only discrete logic chips, the use of wires means that any design change will likely require a new circuit board, not just reprogramming the PLDs. The chip makers came to the rescue again by building much larger programmable chips, including *complex programmable logic devices* (CPLDs) and *field-programmable gate arrays* (FPGAs).

A CPLD contains a set of PLD blocks whose inputs and outputs are connected together by a global interconnection matrix. So a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed. A key feature of the CPLD architecture is the arrangement of logic cells on the periphery of a central shared routing resource. CPLDs use EEPROM, SRAM, or Flash memory to hold the interconnect information.

CPLDs contain the equivalent of many PALs linked by programmable interconnections, all in one integrated circuit. CPLDs are equivalent to about 50 typical PLD devices, and can replace thousands, or even hundreds of thousands, of logic gates.

Programming CPLDs depends on the chip and the application. Some CPLDs can be programmed in a PAL programmer, but that gets difficult if the chip has hundreds of pins, or is surface-mounted. Many CPLDs can be programmed over a serial line from a computer. The CPLD contains a circuit that decodes the data stream and configures the CPLD to perform its specified logic function.

A new interface for programming and testing CPLDs is JTAG (an acronym for Joint Test Action Group). This interface is defined by the IEEE 1149.1 standard for a test access port and boundary scan. Boundary scan is a technique for accessing and stimulating a chip or subsystem via external pins to perform internal test functions on the device. A JTAG interface is primarily used for testing integrated circuits, but it can also be used as a mechanism for debugging embedded systems. A JTAG interface is a special four-pin (data in, data out, clock, test mode select) interface added to a chip. Multiple chips on a board can have their JTAG lines daisy-chained together, so your test probe only needs to connect to a single "JTAG port" to have access to all chips on a circuit board.



Field Programmable Gate Array (FPGA)

While PALs were busy developing into GALs and CPLDs, a separate stream of development was occurring, based on gate-array technology. The resulting device is the field-programmable gate array (FPGA) which was first introduced in the late 1970s. “Field programmable” simply means that the device can be programmed by the user. Many field programmable devices can be programmed with the chip soldered to the circuit board, allowing true “in the field” upgrades to be possible.

FPGAs use a grid of logic gates, similar to that of an ordinary gate array. An FPGA has a collection of simple, configurable logic blocks arranged in an array with interspersed switches that can rearrange the interconnections between the logic blocks. Each logic block is individually programmed to perform a logic function (such as AND, OR, XOR, etc.) and then the switches are programmed to connect the blocks so that the complete logic functions are implemented. FPGAs vary in size from tens of thousands of logic gates to over a million.

The interconnections for the logic blocks are programmable switches. FPGAs may use EEPROM, SRAM, antifuse, or Flash technology to store the programming. In most larger FPGAs the configuration is volatile, and must be re-loaded into the device whenever power is applied or different functionality is required.

FPGAs are typically programmed in hardware description languages (HDLs) like Verilog or VHDL. These high-level languages are used because manual lower level design (such as schematic capture) becomes impractical as designs become large. HDLs also allow the FPGA design to be simulated and tested, prior to implementation in the hardware.

Application Specific Integrated Circuit (ASIC)

ASICs are pretty much what their acronym says - integrated circuits (ICs) designed for specific applications. Unlike standard ICs which are produced by the chip manufacturers, ASICs are designed by the end user and then produced in volume. ASICs allow a user to combine many parts and functions into a single chip, reducing cost and improving reliability.

ASICs can be large or small. They are usually produced in large quantities, and it can be very expensive to produce only a few. If you are manufacturing cell phones, it makes sense to develop an ASIC for your specific needs. If you are flying a space experiment and will need at most a few chips, it would be much more economical to use programmable logic, such as FPGA or CPLD devices.

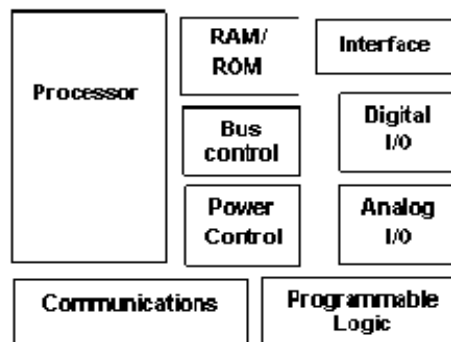
An interesting twist is the production of ASICs that include programmable logic (FPGA, CPLD, PAL) devices as part of the chip. Another very hot technology that combines ASICs with programmable parts is the System-on-chip, described below.

System-on-Chip (SoC)

System-on-Chip combines all the electronics for a complete product into a single chip. SoC's include not only the brains (e.g. microprocessor) but also all required ancillary electronics, such as switches, comparators, resistors, capacitors, timing elements, and digital logic.

SoCs could include:

- Digital/analog functions
- Sensors
- I/O
- Communications
- Ready made sub-circuits (IP)
- Programmable devices
- Digital Signal Processor



SoCs are usually ASICs, though they can be designed to include programmable logic components. SoCs can also be implemented on FPGAs. System-on-chip versions come in several flavors:

Soft Instruction processor architectures allow a designer to customize the CPU architecture. The specific instructions supported, the peripherals available to it and the number of registers are just some ways these devices can be tailored for your application. Some vendors provide mechanisms to add, delete and create highly tailored instructions. Design packages for these architectures sometimes include performance tools with instant feedback on the performance, die size and power requirements of a particular design.

With the final architecture residing in silicon, these types of architectures are well suited for high volume, low cost applications which formerly would have used ASICs.

Configurable processors are FPGA based. In these architectures, standard and customer-derived logic engines can be easily added, modified and extended as needed. By moving discrete logic functionality to internal FPGA the designer gets a highly flexible logic solver, based around a standard processor core. With FPGA logic instead of foundry logic, the logic can be easily revised at any point in the design cycle.

Into the Future

Where is the future taking us? What kinds of new devices and concepts are being considered? Let's take a peak at new ideas being explored.

In-field or reconfigurable SoC

Most system-on-chip (SoC) designs use what is called a platform-based solution, where standard components like a microprocessor core make up a significant portion of the SoC. Custom devices provides further functionality. Some of those devices may be user-configurable (e.g. if a small FPGA or CPLD is part of the System-on-chip device), others may be designer-chosen only. These types of SoC's are usually implemented as ASICs.

A reconfigurable SoC provides the same kind of custom support except that the devices and peripherals are implemented using a reconfigurable matrix. The software must set up the hardware before it can be used. But from that point on, the platform-based SoC software and reconfigurable SoC software will be very similar, assuming that the microprocessor core is the same or similar and the functionality of the peripherals has the same characteristics.

With reconfigurable SoC designs, the hardware functionality can be changed simply by altering the code that performs system initialization. So, SoC could contain an analog-to-digital converter for one application, and then be reconfigured for a digital-to-analog converter, or even a totally different peripheral such as a network device, for another application. Some elements of the reconfiguration can be performed at a later time (after the basic hardware is initialized), allowing software applications to reconfigure devices. Applications that deal with multiple hardware codecs¹ (e.g. streaming multimedia) or encryption methods, for example, could configure devices to the specific codec or encryption method being used at the time, then reconfigure for another codec or method when required for a different data stream.

FPGA microprocessors/systems

¹ codec - Compression-decompression or compressor-decompressor. This may be either a software-only, or a hardware assisted scheme that is used to process files, such as digital video or audio. A CODEC reduces the amount of data to be transmitted by discarding redundant data on the transmitting end and reconstituting the signal on the receiving end.

Some system-on-chip (SoC) devices are implemented entirely on programmable logic, in particular on field programmable gate arrays (FPGAs). Most reconfigurable SoCs fall into this category. However, reconfigurable SoCs use a fixed microprocessor with reconfigurable peripheral devices. What if you could change your microprocessor by just reprogramming the FPGA? What if you could customize the microprocessor for *your* application, then change it when that application changes? That is what the FPGA microprocessor systems offer.

FPGAs have proven themselves capable of handling a wide variety of tasks, from relatively simple control functions to more complex, algorithmic operations. While the time and cost advantages over designing custom ASIC hardware for such functions is well accepted, the advantages of using FPGAs over traditional processors and DSPs for software-oriented applications have been less clear-cut. This is due in large part to a historic disconnect between hardware and software development tools and disciplines.

Recent advances in software-oriented design tools for FPGAs, however, have combined with the ongoing increase in device densities to create a new environment for software developers, one in which the FPGA can be viewed as one possible target (along with traditional and non-traditional processor architectures) for a software compiler. Tools now available can help software engineers make use of FPGA platforms, as well as help these developers take advantage of the high level of algorithmic parallelism that is available when traditional processors (or processor cores) and FPGAs are combined in a single target platform.

FPGA-based computing platforms, particularly those with embedded “soft” microprocessors, have the potential to implement extreme high-performance applications without the up-front risk of creating custom fixed function hardware. Further, by using the latest generation of hardware/software co-design tools it is now possible to use multiple graphical, software-oriented design methods as part of the FPGA design process.

Reconfigurable computing

Someday, perhaps in the not-too-distant future, the computer at your desk may contain a typical microprocessor along with an array of reconfigurable, reprogrammable devices (FPGAs or their successors). Or, the microprocessor may be totally replaced by the FPGAs. As a user, the only thing you’ll note is that your software runs faster, allowing you to get your work done more quickly.

Typical computer systems use a single microprocessor that executes instructions sequentially. They are adaptable and configurable - you can write any kind of operating system or run any sort of application on a microprocessor. However, these systems trade speed for that adaptability.

If you have a fixed set of applications and really need more processing speed, you want an ASIC designed to meet your needs. While you can gain significant improvement in speed, you lose the ability to change the processor/ASIC uses outside of a narrow range of applications. The ASIC

speed increase over general purpose microprocessors comes from a combination of optimization for the specific purpose and the ability to perform processes in parallel.

What if you want speed *and* adaptability? To gain speed, you need to move from the serial processing paradigm to parallel processing. One way to do this is to use multiple processors, each performing operations in parallel. Another way is through reconfigurable computing. Both of these methods keep the adaptability component, allowing the user, through software, to run a wide variety of applications.

To have reconfigurable computing (RC), you need to have hardware that can be reconfigured to implement specific functionality. RC systems contain programmable hardware and may be combined with traditional microprocessors in order to take advantage of the strengths of each device. RC has been used in applications ranging from embedded systems to high performance computing.

Reconfigurable computing uses in-situ reconfigurable FPGAs as computing devices to accelerate operations which otherwise would be performed by software. The FPGA can be programmed with a digital circuit which implements the function to be performed, such as a fast square root operation. The processor can then access this function, as if it were in its own instruction set. When the processor needs another function, such as multiplying two numbers, the FPGA can be reprogrammed for that function.

To make this all work, the FPGA must be capable of being reconfigured quickly and allow only parts of the device to be reprogrammed. Reconfiguration has to be fast, or you quickly eat up the speed advantage you gain from moving the functions from the microprocessor to dedicated hardware. You would also lose too much time if the FPGA had to be entirely reprogrammed when you just want to change part of it. Fortunately, modern FPGAs are up to the challenge.

Reconfigurable computers already exist commercially. Early reconfigurable computers were expensive complicated monolithic FPGA arrays, but most modern commercial and research systems have evolved into relatively less expensive workstation accelerators. Research efforts in academic institutions are considering the establishment and management of parallel reconfigurable computing clusters and high-throughput networks of reconfigurable computers (NORCs). All these individual efforts are creating a new direction - reconfigurable supercomputing.

Summary

Programmable and designable electronics has grown over the years, both in number of devices and in the complexity of the devices. The devices can be roughly grouped by function and complexity.

- Simple, non-programmable logic - ICs
- Simple, programmable logic - PAL, GAL, PLA
- Complex, programmable logic - CPLD, FPGA, reconfigurable computing
- Complex, designable logic - ASIC

NASA Complex Electronics Guidebook for Assurance Professionals

- Complex, designable and/or programmable - SoC

To explore the complex devices in more depth, check out the descriptions in Appendix B.

4 Design Process

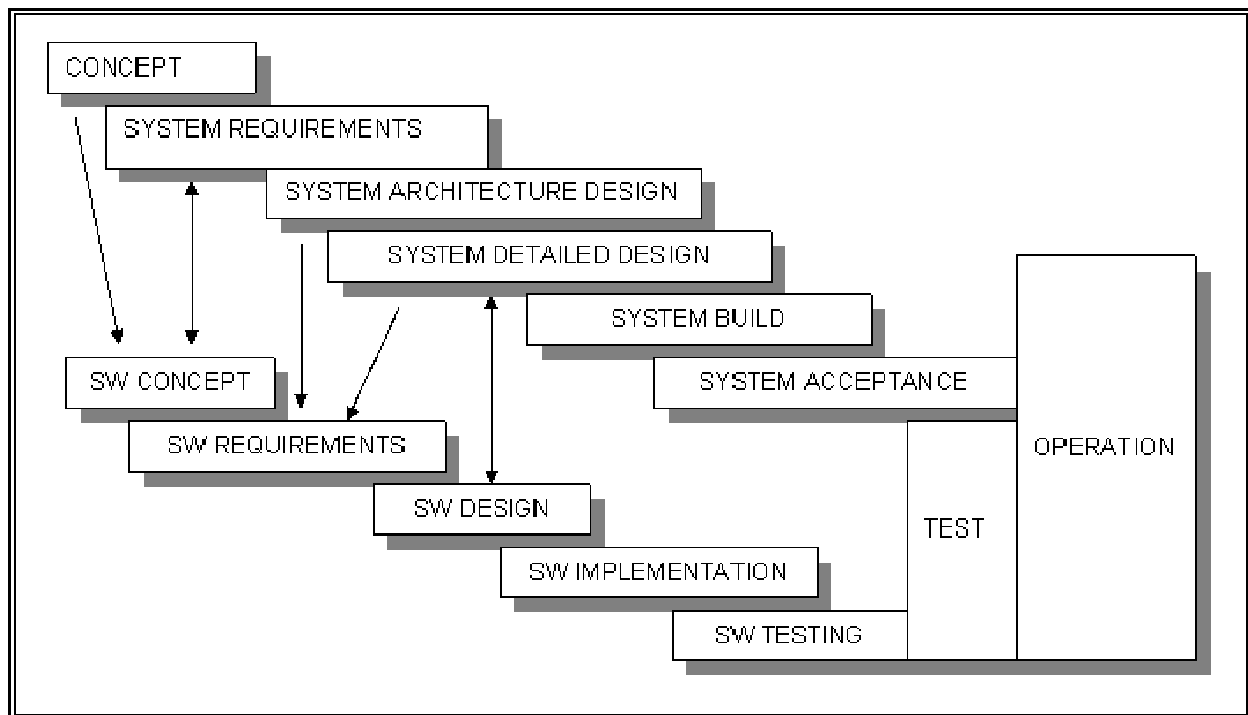
4.1 *Overview of the Complex Electronics Design Process*

Creating complex electronics begins where all systems and subsystems begin - with defining the requirements for the device. Without good requirements, the most elegant design or implementation could utterly fail to meet the original need. Designing and implementing complex electronics occurs within the context of the larger system, as shown in the stylized diagram below.

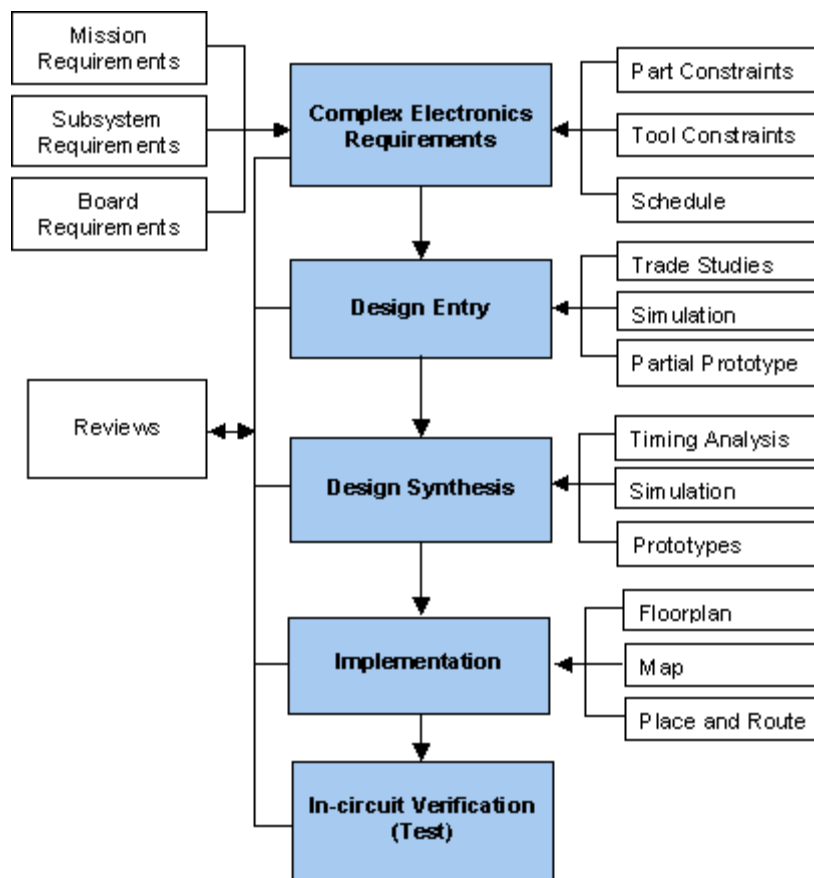
Requirements for the complex electronics are driven by the system they are a part of and the environment where they will be used. A simple home appliance will place fewer demands (requirements) on a device than a sophisticated satellite application will. At a minimum, the home appliance usually doesn't need to worry about high energy cosmic particles that can cause single event upsets. Because these devices are hardware, the process of complex electronics design involves looking at both the chip capabilities and constraints (e.g. how many gates does it have, how much power does it need) and how the design works with and against those constraints and capabilities.

Design Life Cycle

In typical software design, the software requirements are flowed down from the system requirements. Software development may follow a waterfall, iterative, evolutionary, spiral, or other development methodology. Regardless of the development (design) life cycle, the processes of determining the requirements, creating the design, implementing the design, and verifying the implementation are all included. Since it is easy to show graphically, this course will use the waterfall life cycle as a generic life cycle.



Like software, the design and development life cycle for complex electronics can follow any life cycle methodology. Some of the steps vary from those familiar to software developers. Let's look under the hood at the design process for complex electronics.



The basic design flow starts out very similar to software, with the decomposition of system or sub-system requirements to the particular complex electronic device. After that is completed, the engineers take the requirements and generate a design, often in a hardware description language (design entry). The design has to be “compiled” for the device (design synthesis). Synthesis is more complicated than just running a compiler, however. During synthesis, the design is mapped to the logic gates of the device. Simulations are used to verify that the design is correct and can meet the requirements and performance goals.

The implementation of complex electronics involves one more level in the mapping of the logic (design) to the chip. The placement of the logic blocks within the chip, and the routing between blocks, are some of the processes that occur during implementation. This process is loosely comparable to the linking step in software, where the compiled program is fixed up for the software environment it will operate in. At the end of the implementation phase, the final step is to “burn” or program the device.

While the simulation that occurs before the design is committed to hardware can find most defects, the actual hardware device needs to be tested in the circuit. Real signals are applied and the real output is tested. You usually can’t get the degree of testing with in-circuit verification that you can with simulation, because inputting out-of-range signals might be difficult, access to the hardware pins might not be possible, and in real projects, someone always wants to use the hardware as soon as it’s completed. However, functional testing in a variety of conditions is an important verification step. Errors in the silicon chip are possible. Errors induced by the tools are

NASA Complex Electronics Guidebook for Assurance Professionals

more likely. And sometimes the real world acts differently than expected and can influence how the device works.

During this process, what are the assurance roles? Unfortunately, in a lot of designs, the assurance engineers (quality/hardware) only take a look at the system at a high level, and then verify that the final device matches the design and that it was programmed according to a defined process. Many quality assurance engineers do not have the in-depth knowledge required to review a hardware description language design. NASA is looking at how to adequately verify the complex electronics device. More information is provided in Section 6.2.

Here's a quick comparison of the development process for software and complex electronics:

Software		Complex Electronics	
Requirements	Software requirements flow down from system and sub-system requirements	Requirements	Requirements for complex electronics flow down from system and sub-system requirements
Design	Architectural and detailed designs are created, using UML, flow diagrams, and other tools	Design Entry	The design is created primarily in a hardware description language, such as Verilog or VHDL
		Synthesis	Synthesis is the process that takes the higher level designs and optimally translates them to a gate-level design which can be mapped to the logic blocks in a complex electronic device.
Code	The design is translated (manually or automatically) into a programming language (code), and then compiled into an executable module	Implementation	Implementation is where the design meets the silicon - the mapping created by synthesis is converted into a chip layout. The final step in implementation is to put the design into the chip - either through programming (burning) or manufacturing (for ASICs)
Test	The software is tested in individual units and as part of the system. Testing may involve additional software that simulates inputs to the software under test	Test	Testing occurs during the design entry, synthesis, and implementation phases, in the form of simulations. Both expected (valid) and unexpected inputs are tested. Once the device is created, it is tested as part of its sub-system (in-circuit testing).

4.2 Requirements and Specifications

The first step in the design process is to understand (and document) the functions the complex electronics device must perform and the constraints under which it operates. In the same way that you shouldn't just start writing the software to meet what you think are the requirements, designing complex electronics should start with creating the specification. The act of documenting the requirements has some useful effects that actually can save you time in the long run:

- The design team thinks through the issues and reaches agreement. Some issues are well understood at a high level, but raise additional questions when working at the hardware level.
- Interfaces to other areas (software, other hardware) are defined and available for review by all affected parties.
- Non-engineers can understand what the chip or device is supposed to do.
- If the trade-offs and rationale are documented, as well as the requirements, future design changes will require less impact assessment.
- The requirements can be reviewed to assure that they provide measurable, testable criteria.
- Requirements traceability into the design and implementation can be performed - which is vital in mission- or safety-critical applications.

A good specification for complex electronics will contain:

- A description of how the device fits into the larger system. A block diagram is very helpful.
- A description and list of all the major functions the device will perform. A block and/or flow diagram can be used to show this information.
- A description of the device and interfaces, such as:
 - Chip physical information (size, type, number of pins, etc.)
 - I/O pin mapping and description (output drive capability, input threshold level)
- Timing estimates for:
 - Setup and hold times for input pins
 - Propagation times for output pins
 - Clock cycle time
- High-level estimates and goals
 - Gate count estimate
 - Power consumption target
- Constraints on the device
- Other requirements or criteria the device must implement
- Design-related choices (may be in a management plan)
 - Tools that will be used at all stages of development
 - Hardware Description Language chosen

Assurance Roles

What role does software or hardware quality assurance play in verifying the requirements specification for complex electronics? The reality is that many assurance engineers, regardless of their specialty, have little understanding of the complexities of these devices. Any review or evaluation will be to the level of knowledge of the assurance engineer.

- **Hardware quality assurance engineers** are the primary assurance people to deal with complex electronics. Hardware QA's with a background in electronics will evaluate the requirements for the complex electronic device for accuracy, completeness and compatibility with the rest of the system. When hardware quality assurance has little exposure to, or understanding of, complex electronics, the evaluation will often be very high-level.
- **System safety engineers** will be involved in the review when the devices are part of safety-critical systems or are used as controls or mitigations for hazards. As with hardware quality assurance, system safety engineers usually do not have an in-depth understanding of complex electronics.
- **Software assurance engineers** at NASA are currently only rarely involved with complex electronics. Significant education is required to be able to intelligently review the requirements and specification for complex electronics. However, after taking this course, you should be able to review specifications for complex electronics at a high-level and look for:
 - Problems with interfaces to other system elements or to the software running on the system
 - Problems, issues, or concerns regarding the functions that are implemented in the hardware
 - Additional constraints that may not be included in the specification, or incorrect constraints
 - Areas where software functions could be implemented in the complex electronics

4.3 Design Entry

The first step in creating a design for complex electronics is to choose how you will enter (capture) your design. Early chip designs were primarily performed with schematic capture. Schematic capture (also called schematic entry) creates the electronic diagram, or schematic, of the electronic circuit. This is usually done interactively with the help of a schematic capture tool also known as schematic editor.

While schematic capture works fine for simple designs, complex electronics almost always require the use of a hardware description language (HDL). HDLs are any languages that are used for formal description of electronic circuits. These languages can describe the operation, design, and simulation tests of the circuit. HDLs can show several aspects of the design, including the temporal behavior and spatial structure. One major difference between HDLs and software languages is the aspect of timing and concurrency.

One very nice aspect of HDLs is that they can be used as “executable specification” to simulate the circuit. Simulation software can be part of the tool suite provided by the vendor or a third-

party program. Simulators read the HDL “code” and model the structure and flow of the circuit through time.

The two primary description languages are VHDL and Verilog. The next section (Hardware Description Languages) will discuss these two languages in greater detail. Older HDLs, such as ABEL and CUPL, are still in use, especially for simple designs. Another trend in hardware description languages is to add hardware-specific elements to software programming languages. JHDL is implemented on top of the Java language. SystemC adds hardware constructs as a C++ class library. Still, VHDL and Verilog are by far the most common hardware description languages in use.

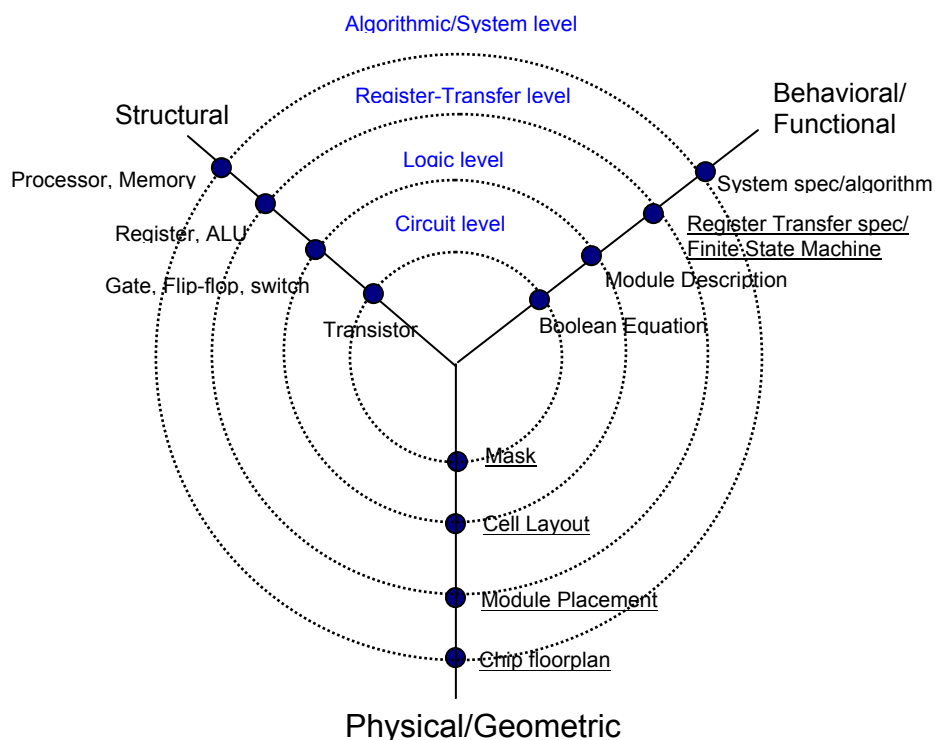
Regardless of the method chosen to input the design (a hardware description language or schematic capture), a software tool (or tool suite) is required. Unlike most software development efforts, where tools other than editors, compilers, development environments, and version management software are rarely used, electronics designers require, and use, fairly sophisticated tools. All major complex electronics vendors offer design tools optimized for their devices at a relatively low cost. Third-party tools are common, and can provide additional capability. These tools are also often quite expensive. However, because the boundaries between design entry, simulation, synthesis, and place-and-route are well defined, the designer can mix-and-match tools from different vendors.

A tool suite may include the following types of tools:

- HDL capture and design environment
 - May include configuration management
- HDL simulator
- Logic Analyzer
- Logic Synthesis (this is a critically important tool)
- Layout (physical synthesis)
- Design management

Design Views

Complex electronic devices are designed at several levels, and with several “views”, or ways of looking at the device. Software shares some of these views (e.g. the behavioral/functional view and the structural view), though software isn’t concerned with physical layouts. Each of the various views of the device are refined at each of the levels of representation. The Y diagram below shows how all these views and levels are related.



Modern design approaches for complex electronics focuses on the behavioral/functional aspects of the devices, and uses sophisticated tools to create the appropriate structural and physical aspects of the design. Earlier design approaches required much more manipulation at lower levels of the device circuit. With increasing complexity of the devices, the design aspects have been moved “up” into a more abstract domain, and the grunt work of converting the design into a usable circuit is left to the tools. This abstraction allows the designer and others to understand how the device functions within the context of the system.

A specification in a hardware description language consists of one or more modules. The top level module specifies a closed system containing both test data and hardware models. Component modules normally have input and output ports. Events on the input ports cause changes on the outputs. Events can be either changes in the values of wire variables (i.e., combinational variables) or in the values of reg variables (i.e., register variables), or can be explicitly generated abstract events. Modules can represent pieces of hardware ranging from simple gates to complete systems (e.g., microprocessors), and they can be specified either behaviorally or structurally, or by a combination of the two.

A **behavioral specification** defines the behavior of a digital system (module) using traditional programming language constructs (e. g., ifs, assignment statements). This description of a complex electronic device divides the device (chip) into several functional blocks that are interconnected. A hardware description language (HDL) is used to describe the behavior of each block. Functional blocks can be a finite state machine, a set of registers and transfer functions, or even a set of other interconnected functional blocks.

A **structural specification** expresses the behavior of a digital system (module) as a hierarchical interconnection of sub modules. The components at the bottom of the hierarchy are either primitives or are specified behaviorally. It is in the structural specification that individual inputs and outputs are defined.

Assurance Roles

At the design entry stage,

- **Hardware quality assurance engineers** with a background in electronics will, ideally, evaluate the design for the complex electronic device against the requirements. For many projects, especially when hardware quality assurance has little exposure to, or understanding of, complex electronics, no evaluation will be performed.
- **System safety engineers** will review the design of the devices when they are part of safety-critical systems. Since few system safety engineers are experts in complex electronics, they will work with the designer or hardware QA to evaluate the design from a safety perspective.
- **Software assurance engineers** are not currently involved. However, after taking this course, you should be able to provide a cursory review of the design, especially the VHDL or Verilog code.

4.3.1 Abstraction and Modeling

Hardware description languages can be used to describe complex electronics at many different levels of abstraction. An abstraction is a simplified representation of something that is potentially quite complex. It is often not necessary to know the exact details of how something works, is represented or is implemented, because we can still make use of it in its simplified form.

The levels of abstraction for a complex electronic device are:

- System or Behavioral
- Algorithm
- Register-Transfer Level (RTL)
- Gate

The highest level of abstraction is the system level, where the device is mostly a black box that interacts with its environment. Very little is known about the internals of the device, but you do know how it functions (its behavior).

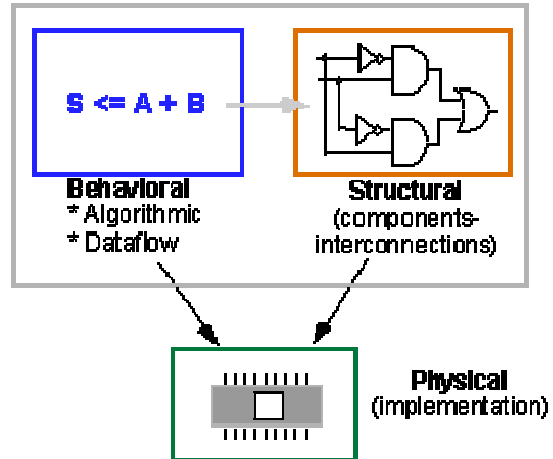
A pure algorithm consists of a set of instructions that are executed in sequence to perform some task. A pure algorithm has neither a *clock* nor detailed delays. Some aspects of timing can be inferred from the partial ordering of operations within the algorithm. The algorithmic level of abstraction is similar to software programming (“While ready, do task A and task B, then do task C). Because of the lack of timing information, this level is not synthesizable (able to be mapped to hardware).

The Register-Transfer Level (RTL) description has an explicit clock. All operations are scheduled to occur in specific clock cycles, but there are no detailed delays below the cycle level. A single global clock is not required but may be preferred. In addition, retiming is a feature that allows operations to be re-scheduled across clock cycles. The RTL level is the input to the synthesis tool.

The gate level of abstraction is the output from the synthesis tool. A gate level description consists of a network of gates and registers, along with technology-specific delay information for each gate.

A complex electronics device can be described in one of three domains:

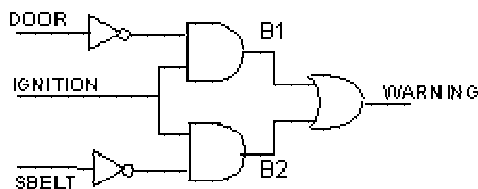
- Behavioral
- Structural
- Physical



Hardware description languages deal with the first two (behavioral and structural). The mapping from the behavioral and structural domains to the physical implementation are performed by the synthesis and place-and-route tools.

As an example, let us consider a simple circuit that warns car passengers when the door is open or the seatbelt is not used whenever the car key is inserted in the ignition lock. At the behavioral level this could be expressed as,

$$\text{Warning} = \text{Ignition_on} \text{ AND } (\text{Door_open} \text{ OR } \text{Seatbelt_off})$$



The **structural** level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is usually closer to the physical realization of a system.

4.4 Hardware Description Languages

4.4.1 Overview of Hardware Description Languages

While schematic capture works well for small circuits and devices, complex designs require the ability to abstract at a higher level. Thus, hardware description languages (HDLs) were born. One difference between HDLs and software languages is that HDLs are essentially *models* of the hardware. The languages were initially created to allow simulation of the design, and contain all the necessary capabilities to create test benches and simulation models. Simulation of the complex electronics is *very* common in the design community.

There are two major HDLs that are currently in use: Verilog and VHDL. This course will provide you with a cursory overview of these are the two languages. However, each of the languages is a course (or two) in their own right. Several good tutorials on the languages are provided on the Links page.

The Verilog hardware description language was invented by Philip Moorbyin in 1983. The first Verilog synthesis tool was introduced in 1987. Verilog was placed in the public domain, and is now specified by an IEEE standard (IEEE 1364). This language enables specification of a digital system at a range of levels of abstraction, such as switches, gates, Register-Transfer Layer (RTL), and higher.

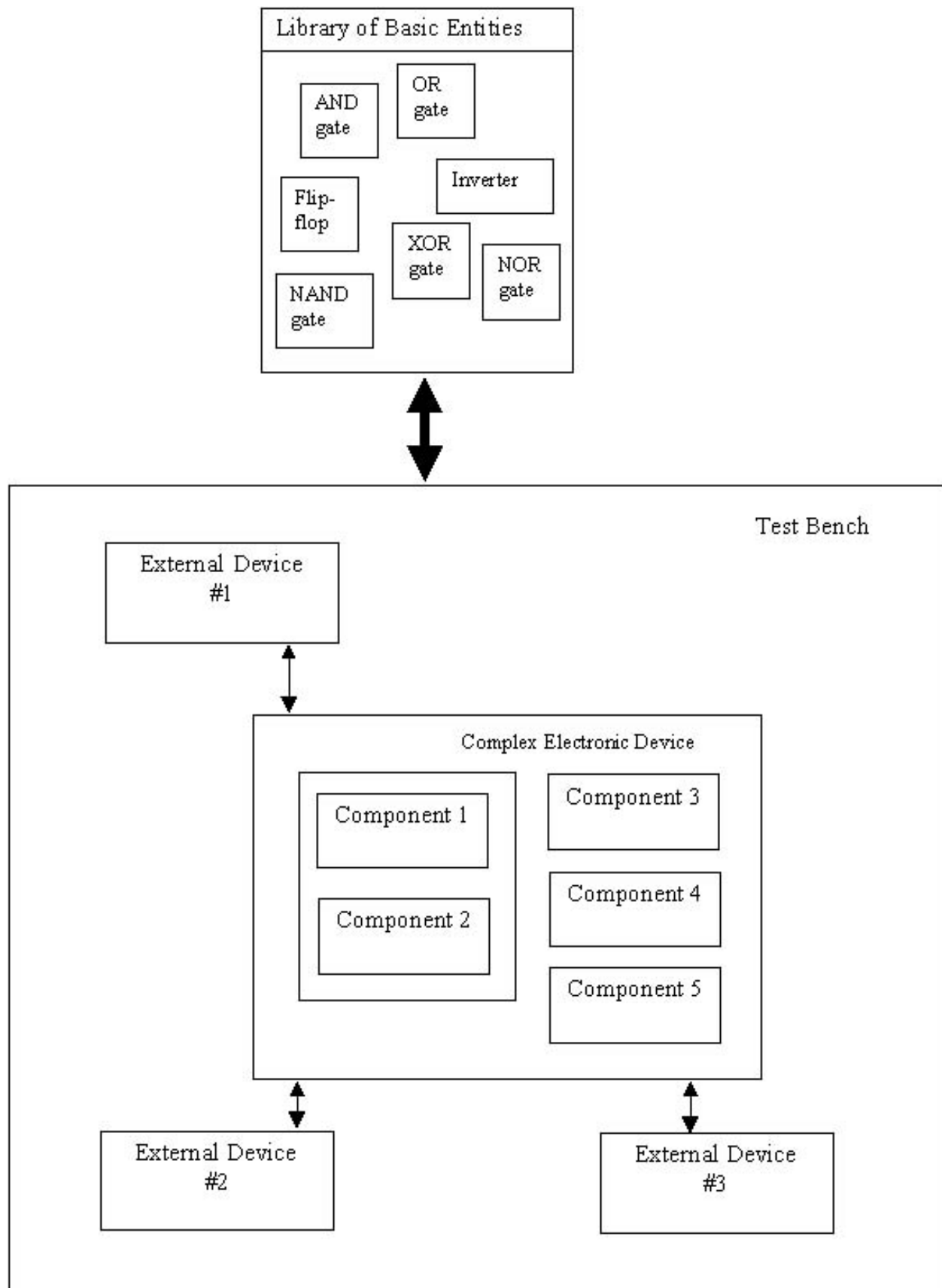
VHDL is a “double acronym”, and stands for “VHSIC Hardware Description Language”. VHSIC is “Very High Speed Integrated Circuit”. Just call it VHDL and don’t worry about the acronym. VHDL is also specified by an IEEE standard (IEEE 1076). VHDL was developed over time, culminating in its initial release in 1987, with an update in 1993.

General Hardware Description Language Concepts

As you learn about HDLs, there are a few major differences from software languages you need to keep in mind. First, software is inherently sequential - one instruction is executed after another. Even in multi-threaded or multi-tasking systems, no two tasks operate at the exact same moment! Hardware, however, is parallel in nature - multiple events can be happening simultaneously. Hardware description languages have ways to describe concurrency (parallel execution) and to specify timing. Second, HDLs describe hardware. While at the highest abstraction an HDL can define an algorithm similarly to a software language, at the lower levels of abstraction that algorithm is translated into gates and I/O.

Hardware description languages model two aspects of the hardware: structure and behavior. These two aspects are independent - the structure of the hardware is not dependent on the behavior, and vice versa. The interfaces (input/output signals) from the device to the outside world is part of both the structure (what the device is made of) and the behavior (what it does with the signals). In addition, because HDLs were originally designed as simulation languages, they can create test benches to exercise and test the device with simulated “real world” devices.

The first step when designing and modeling complex electronics in a hardware description language is to partition the design into natural abstract blocks, known as components. Each component is the instantiation of a design entity, which you normally model in a separate system file for easy management and individual compilation by simulation or synthesis tools. You then model the total system using a hierarchy of components, known as a design hierarchy, which consists of individual subcomponents (subdesign entities) brought together in one higher level component (design entity). In other words, you start with very simple entities (e.g., AND gate) and put them together into components (logical subdivisions within your device), which together become the model of your device.



NASA Complex Electronics Guidebook for Assurance Professionals

The two main elements of the HDL description of the complex electronic device are the *architecture body* (the structure) and the *behavioral architecture*. The architecture body describes the implementation of a module's inputs and/or outputs. The electrical values of the outputs are some function of the values of the inputs. Of course, each module can be made up of sub-modules, down to the basic entities. The connections between the sub-modules (inputs/outputs) are made using signals.

The architecture body contains:

- Signal declarations, for internal interconnections
- Entity ports (also treated as signals)
- Component instances (instances of previously declared entity/architecture pairs)
- Port maps in component instances (connect signals to component ports)
- Wait statements

The behavioral architecture describes the algorithm performed by the module. While the architecture body described the inputs and outputs, the behavioral architecture describes what goes on to convert those inputs to outputs. More complex behaviors cannot be described purely as a function of inputs. In systems with feedback, the outputs are also a function of time. Fortunately, hardware description languages provide features to handle time as part of the behavior.

The behavioral architecture contains:

- process statements
- sequential statements
- signal assignment statements
- wait statements

You can describe the behavior of a module without describing its structure. You might want to do this if you have an off-the-shelf component as part of your design. You don't really care about the internal structure of the component; you just want to describe what it does.

This course won't go into significant detail on the two main hardware description languages (VHDL and Verilog). However, it would be good for you to have at least an overview of the languages. The links below will take you to some excellent tutorials on VHDL or Verilog.

VHDL Tutorials

http://www.seas.upenn.edu/~ese201/vhdl/vhdl_primer.html

<http://www.gmvhdl.com/VHDL.html>

Verilog Tutorials

<http://www.asic-world.com/verilog>

http://ca.olin.edu/cawiki/Fall_2006/Materials?action=AttachFile&do=get&target=VerilogTutorial.pdf

http://www.doulos.com/knowhow/verilog_designers_guide/

Comparison of VHDL and Verilog

You can design complex electronics in either of the main hardware description languages (VHDL and Verilog). Both provide all the capabilities you require. While the choice of language is mostly a personal preference, there are some differences between the two that may be important for specific applications. Here's how the two compare:

	VHDL	Verilog
Similarity to software programming language	Pascal and Ada	C
Level of abstraction	VHDL models well from the system level down to the RTL level, with some modeling at the gate level.	Verilog has less system modeling capabilities than VHDL, but more capabilities at the gate level.
Compilation	Allows separate compilation of multiple design-units (entity/architecture pairs) that reside in the same system file.	With Verilog, care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.
Data types	VHDL has a multitude of language or user defined data types that can be used. As a result, dedicated conversion functions are needed to convert objects from one type to another.	Verilog data types are very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. All data types used in a Verilog model are defined by the Verilog language and not by the user.
Design reusability	Procedures and functions may be placed in a package so that they are available to any design-unit that wishes to use them.	There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module.

	VHDL	Verilog
Ease of learning	VHDL may seem less intuitive at first for two primary reasons. First, it is very strongly typed; a feature that makes it robust and powerful for the advanced user after a longer learning phase. Second, there are many ways to model the same circuit, specially those with large hierarchical structures.	Probably easiest to learn with no prior exposure or knowledge.
High level constructs	<p>VHDL contains more constructs and features for high-level modeling than Verilog. Abstract data types can be used along with the following statements:</p> <ul style="list-style-type: none"> • package statements for model reuse • configuration statements for configuring design structure • generate statements for replicating structure • generic statements for generic models that can be individually characterized 	<p>Verilog has no high-level modeling statements similar to VHDL's. Verilog allows you to parameterize models by overloading parameter constants.</p>
Language extensions	VHDL allows architectures and subprograms to be modeled in another language by using the "foreign" attribute.	The Programming Language Interface (PLI) is an interface mechanism between Verilog models and Verilog software tools.
Libraries	VHDL uses a library to store compiled entities, architectures, packages and configurations. Useful for managing multiple design projects	There is no concept of a library in Verilog. This is due to it's origins as an interpretive language.
Low level constructs	Simple two input logical operators are built into the language, they are: NOT, AND, OR, NAND, NOR, XOR and XNOR. Any timing must be separately specified.	The Verilog language was originally developed with gate level modeling in mind, and so has very good constructs for modeling at this level and for modeling the cell primitives of ASIC and FPGA libraries.

	VHDL	Verilog
Managing large designs	Configuration, generate, generic and package statements all help manage large design structures.	There are no statements in Verilog that help manage large designs.
Operators	Similar operators to Verilog with the addition of a mod operator.	Similar operators to VHDL with the addition of a unary reduction operator.
Parameterizable models	A specific bit width model can be instantiated from a generic n-bit model using the generic statement.	A specific bit model can be instantiated from a generic n-bit model using overloaded parameter values.
Procedures and tasks	Allows concurrent procedure calls.	Does not allow concurrent task calls.
Readability (This is more a matter of coding style and experience than language feature)	VHDL is a concise and verbose language; its roots are based on Ada.	Verilog is more like C because it's constructs are based approximately 50% on C and 50% on Ada.
Structural replication	The <i>generate</i> statement replicates a number of instances of the same design-unit or some sub part of a design, and connects it appropriately.	There is no equivalent to the <i>generate</i> statement in Verilog.
Test harnesses	VHDL has generic and configuration statements that are useful in test harnesses.	Verilog does not have similar statements.
Verboseness	Because VHDL is a very strongly typed language, models must be coded precisely with defined and matching data types. Models are often more verbose, and the code often longer, than it's Verilog equivalent.	Verilog allows signals representing objects of different bit-widths to be assigned to each other. The signal representing the smaller number of bits is automatically padded out to that of the larger number of bits. This has the advantage of not needing to model quite so explicitly as in VHDL, but does mean unintended modeling errors will not be identified by an analyzer.

Coding Standards

Just as in writing software for embedded applications, a coding standard is important when more than one person will ever have to use the source code. The big danger is that when the person who wrote the original code leaves or moves on to another project, no one will understand how it

works if the code ever has to change. Even the original designer is likely to forget it in several months.

One can easily write individual lines of understandable HDL code that collectively become extremely difficult to follow. A good coding standard will help alleviate this by providing guidelines for hierarchical structures and component instantiations. For instance, many books use various types of flip-flops as examples to model component instantiations (mostly because these are already understood by the readers). However, in practice, it's generally poor coding style to instantiate logic by mapping each register to various kinds of flip-flops. This can lead to longer, more obfuscating logic that does not take advantage of the ability to write in VHDL and Verilog at a higher level.

The important factors in a HDL coding standard are:

- Consistent and defined style
- Guidelines on writing understandable code
- Commenting guidelines
- Information to capture in comments at each level
- Naming convention (for consistency)

4.4.2 Programming Example

Now that you have learned a bit about complex electronics, and the languages used to create the design, some examples are in order. These examples are *very* simple. However, they will hopefully help familiarize you with VHDL and Verilog programming constructs.

Before we begin, let's review some basic ideas of VHDL and Verilog.

VHDL uses the concept of a “design entity”, which consists of two design units. The *entity declaration* defines the external interface. The *architecture body* details the internal structure, and can define the entity's behavior, structure, or both.

Verilog uses the concept of a “module” rather than “entity”. Like VHDL, the port declarations (external interface) are separate from the *module body*, which defines the internal behavior and/or structure.

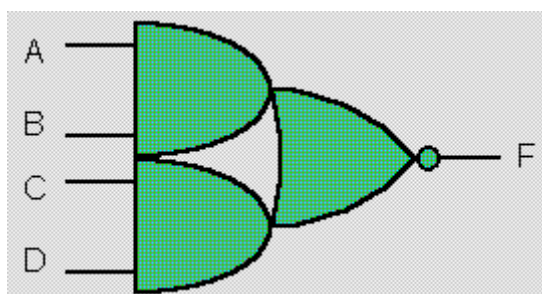
Example 1: And-or-invert (AOI) gate

Description: This gate takes two sets of signals, each of which is ANDed together, ORs the resulting signals, and finally inverts the results.

Truth table (not complete):

A	B	A&B	C	D	C&D	OR	Result (F)
1	1	1	1	1	1	1	0

0	1	0	1	1	1	1	0
1	0	0	1	1	1	1	0
1	1	1	0	1	0	1	0
1	1	1	1	0	0	1	0
1	1	1	0	0	0	1	0
1	0	0	1	0	0	0	1
0	0	0	0	0	0	0	1
0	0	0	1	1	1	1	0

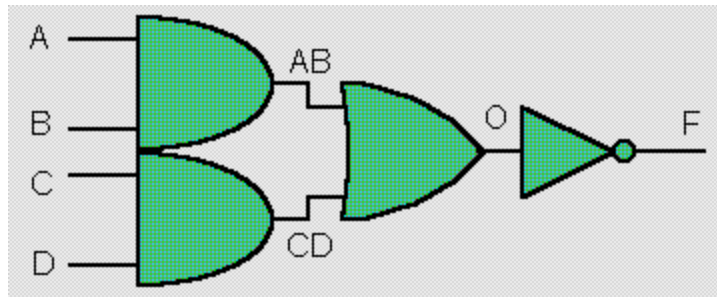


VDHL Code for AOI gate	Verilog Code for AOI gate
<pre>-- VHDL code for AND-OR-INVERT gate library IEEE; use IEEE.STD_LOGIC_1164.all; entity AOI is port (A, B, C, D: in STD_LOGIC; F : out STD_LOGIC); end AOI; architecture V1 of AOI is begin F <= not ((A and B) or (C and D)); end V1; -- end of VHDL code</pre>	<pre>// Verilog code for AND-OR-INVERT gate module AOI (A, B, C, D, F); input A, B, C, D; output F; assign F = ~((A & B) (C & D)); endmodule // end of Verilog code</pre>
Explanation of VHDL code	Explanation of Verilog code
<ul style="list-style-type: none"> Comments begin with "--" library/use: provides the entity with access to all the names declared in the package STD_LOGIC_1164. "entity AOI is" - starts the entity description and assigns it the name AOI. "port {...}" specifies the input and output signals and the data type of each "end AOI;" terminates the entity declaration "architecture V1;" gives a label (V1) to the architecture body and connects it to the AOI entity declaration "begin" starts the architecture statement (the details). 	<ul style="list-style-type: none"> Comments begin with "//" "module AOI (A, B, C, D, F);" - starts the module description and assigns the module a name (AOI). "input A, B, C, D; output F;" - declares which of the signals are inputs and which are outputs. "assign F = ~((A & B) (C & D));" - logic statement defining the behavior of the signals. The concurrent assignment executes whenever one of the four ports A, B, C or D change value. The ~, & and symbols represent the bit-wise not, and and or operators respectively. "endmodule" - terminates the module definition

- “F <= not ((A and B) or (C and D));” specifies the behavior of the signals
- “end V1” terminates the architecture body.

Example 2: AOI Gate with internal signals and timing

Description: This gate is the same as the first example, except that the internal signals (between the AND, OR, and NOT gates) are explicitly identified. Additionally, timing delays are included in this example.

**VHDL code**

```
library IEEE; use
IEEE.STD_LOGIC_1164.all; entity AOI is
port (A, B, C, D: in STD_LOGIC; F : out
STD_LOGIC); end AOI; architecture V2 of
AOI is signal AB, CD, O: STD_LOGIC;
begin AB <= A and B after 2 NS; CD <= C
and D after 2 NS; O <= AB or CD after 2
NS; F <= not O after 1 NS; end V2;
```

Explanation of VHDL code

The majority of the statements are exactly the same as in the previous example. In the architecture body, the internal signal behavior is defined.

Whenever a signal on the right side of the assignment (e.g. “A and B”) is evaluated whenever either A or B changes value. The signal on the left side of the assignment (e.g. “AB”) is updated with the new result after a delay of 2 nanoseconds.

In this example, if port A changed value, the result would propagate through the entity, to the final output, with a total delay of 5

Verilog code

```
// Verilog code for AND-OR-INVERT gate
module AOI (A, B, C, D, F); input A, B, C,
D; output F; wire F; // the default wire AB,
CD, O; // necessary assign AB = A & B;
assign CD = C & D; assign O = AB | CD;
assign F = ~O; endmodule // end of Verilog
code
```

Explanation of Verilog code

In Verilog, a wire represents an electrical connection. A wire declaration looks like a port declaration, with a type (wire), an optional vector width and a name or list of names. “wire AB, CD, O;” declares three wires (the internal signals).

The assign statements (e.g. “assign AB = A & B;” are the same format as in the previous example. They break out the logic from one statement into several, using the internal signals (wires). These statements are independent and executed concurrently, and are not necessarily executed in the order in which they are written.

nanoseconds.	
--------------	--

4.5 Synthesis

Design synthesis is the process that takes the higher level designs and optimally translates them to a gate-level design which can be mapped to the logic blocks in a complex electronic device. It is during synthesis that timing and area constraints can be specified by the user. Unlike software which executes sequentially, the elements of a complex electronic chip will execute in parallel, with specific timing requirements. However, in general, you can think of synthesis as a form of compiling, translating the readable language into instructions that are implemented in the integrated circuit.

The synthesis step transforms the behavioral and structural specifications into an optimized netlist of gates. The netlist is just a description of the various logic gates in the design and how they are interconnected. During synthesis, the designer can optimize parameters and constraints in the final chip. For example, a certain amount of delay may be necessary when accessing an outside element like a sensor. This delay can be included as a constraint during the synthesis process. Other constraints may be power consumption and signal timing.

Synthesis is performed almost exclusively by a software tool. Modern synthesis tools do an excellent job optimizing complex designs, so designers do not need to manually perform that task. However, user input to the tools does have an effect on the output. For example, synthesis tools behave very differently given a common set of constraints. These timing-driven tools perform complex trade-offs to achieve the timing constraint you specify, including adding extra parallel logic to paths where there is negative timing slack, or optimizing a critical path at the expense of a non-critical one. When you over-constrain a design, the tool sees many, many paths that don't meet timing and can generate lots of extra logic in a futile attempt to make all of them hit the timing goals. This can result in a much larger design with reduced overall timing performance. In a timing-driven tool the idea is to give the tool the real timing specifications, and let it work to meet that goal. Once that performance has been met, the tool will start optimizing for less area which translates to cost savings in your device. This can produce an even faster design because routing delays can be reduced by having less logic in non-timing-critical areas.

Simulation

Simulation is used in the design of complex electronics at several levels. One very nice aspect of hardware description languages is that they are “executable”, and simulators that can run the code are very common. Simulators are usually part of the tool suite provided by the vendor of the complex electronic device (e.g., FPGA).

After design entry, the design is simulated at the register-transfer level (i.e., the HDL code). Simulation at this level is very fast, allowing the designer to implement many simulations to

fully understand how the device will operate. Simulation can be used to help optimize the design and refine the logic, though designers need to be careful not to use it in the undisciplined software-style code-and-fix mode. Simulation of the HDL code will look at signals and variables to check their value, trace functions and procedures, and use breakpoints to check the status of the device at specific events. This process is very similar to using a software debugger. One caveat with simulation at this level of design is that some properties are not yet defined, such as timing and resource usage.

After design synthesis, but before physical implementation, functional simulation is used to help verify the design. The goal of functional simulation is to ensure that the logic of the design does what you want it to do, per the specification, and that it produces the correct results. This type of simulation is very important to get as many bugs out of the device as possible. If any errors are discovered, then the *design entry* step is re-visited and necessary required changes are made, leading to a successful simulation.

After the design has been implemented, but before the device is actually programmed, a final simulation with full timing information is performed. The placement and routing process will determine any delays and other timing information, which are back-annotated to the gate-level netlist. This simulation is a much longer process, because of the timing aspects. A static timing analysis might be substituted for the timing simulation. Static timing analysis calculates the timing of combinational paths between registers and compares it against the designer's timing constraints.

Test Benches

Test benches for complex electronics are not made of wood or metal, but of Verilog or VHDL code. They are special programs designed to test your complex electronics design. While simulators can verify simple designs, more complex designs require a test bench to adequately verify the design.

A Test Bench is a HDL design you create which can load your circuit, apply stimulus to its inputs (including defining multiple clocks) and check the outputs for correctness. Because it is a program you write, you have control over how your circuit is built and simulated. In addition to the above capabilities, a test bench can provide behavioral or structural models for everything on the PC board. In this way, it enables you to simulate the entire system including your complex electronics design(s) as well as external bus interfaces, external memories, etc. An engineer can design the test benches to automatically check important data conditions and to report any errors to a command window.

Comprehensive, up-front verification is critical to the success of a design project, and test benches should be created as you start to design your device. A HDL test bench/simulator can become your primary design development tool. When simulation is used right at the start of the project, you will have a much easier time with synthesis, and you will spend far less time re-running time-intensive processes, such as place-and-route tools and other synthesis-related software.

Test benches can be quite simple, applying a sequence of inputs to the circuit over time. They can also be quite complex, perhaps even reading test data from a disk file and writing test results to the screen and to a report file. A comprehensive test bench can, in fact, be more complex and lengthy (and take longer to develop) than the circuit being tested.

Depending on your needs (and whether timing information related to your target device technology is available), you may develop one or more test benches to:

- Verify the design functionally (with no delays).
- Check your assumptions about timing relationships (using estimates or unit delays).
- Simulate with annotated post-route timing information so you can verify that your circuit will operate in-system at speed.

A typical VHDL or Verilog test bench is composed of three main elements:

- **Stimulus Generator** - drives the unit under test with certain signal conditions (correct and incorrect values, minimum and maximum delays, fault conditions, etc.)
- **Unit Under Test** - represents the device undergoing test or verification
- **Verifier** - automatically checks and reports any errors encountered during the simulation run. Compares model responses with the expected results.

Test benches are created by human beings, often by the designer, and are subject to faults and failings like any human endeavor. If the logic of the test bench is incorrect, or if a particular stimulus is not defined, then the end result of the tests may not show an actual error. It's something to keep in mind if you are called on to review verifications for a piece of complex electronics. You can't assume that the test bench accurately and completely tested the device - especially if the device will be used in a safety-critical application.

Assurance Roles

At the design synthesis stage, assurance and safety engineers are usually not involved. A hardware assurance engineer might participate in or witness simulations of the device, or assess the test bench created for the simulation. The simulation results may be reviewed by the system safety engineer.

Ideally, an assurance engineer should review the constraints used in the synthesis process and assess the simulations that are performed. The test bench should also be assessed, to verify that it is correctly testing the device being created. All of these activities require a knowledgeable engineer who has experience with complex electronics.

A software engineer taking this course should be able to follow along with any simulations that are performed, and be able to assess if the results match the interface specifications (e.g. if the output on a particular pin is within the valid range).

4.6 Implementation

Once a design has been created, simulated, and synthesized, the next step is implementation of the design in the particular complex electronic device. In software, implementation is usually translating the design into source code and compiling it. In complex electronics design, implementation is where the design meets the silicon - the higher level design is converted into a chip layout.

The implementation process uses the tools supplied by the device (e.g., FPGA) vendor. The functions that were defined in the design have to be matched to the available blocks, gates, and other logic elements on the chip. Some basic steps in implementing a design are:

- Floorplan
- Translate
- Map
- Place and Route

The exact order of a step (or even the name a step/process is given) varies across different groups, companies, and documents. So don't take the information in this course as the absolute right way to do things. However, the concepts presented here are common across the industry and will be implemented to some extent in all programs - perhaps as part of an automated tool or under a different name. Being familiar with the concepts will help you "speak the language" when talking with a design engineer working with complex electronics.

Floorplanning is the process of identifying structures that should be placed close together, and allocating space for them. In designing complex electronics, there are multiple goals that must be met, and the goals often conflict. Finding the best balance between the various goals and requirements is something of an art. Some goals are:

- Minimize space on the chip (allows choice of less costly chips)
- Meet or exceed required performance
- Place everything close to everything else to minimize transmission time in the signal paths

Floorplanning does not have to be performed by the designer for many designs/chips. Most tool suites will perform this step as part of the automated sequence that takes the design and implements it in the chip. However, if you are creating an ASIC, need the absolute best timing possible, or are trying to cram a large design into a not-so-large chip, you will probably need to actively floorplan.

Done correctly, there are no negatives to floorplanning. However, if the floorplanning is done with no regard for the architecture of the chip, then it is possible to actually do a worse job than the automated tool. It is also possible that there are constraints that are not well understood until placement is complete, and routing commences.

As a general rule, data-path sections benefit most from floorplanning, and random logic, state machines, and other non-structured logic can be safely left to the placer section of the place and route software. Data paths are typically the areas of your design where multiple bits are processed in parallel with each bit being modified the same way with maybe some influence from adjacent bits. Example structures that make up data paths are adders, subtractors, counters, registers, and muxes.

Translation involves converting the results of the synthesis process to the format supported internally by the vendor's place-and-route tools. The incoming netlist is checked for adherence to design rules and is then optimized for the chip.

Translation may also be referred to as compilation or compiling. This process is automatic, but it takes some wading through the reports produced by the tool to verify that the translation/compile was correct. An intelligent post-processor, rather than the designer (or worse, the quality assurance engineer), should be used to find syntax and binding errors - otherwise you will have to do this for each design modification!

Mapping takes the logic blocks and determines what logic gates and interconnections on the device should be used to implement those blocks. During the mapping step, the functions within the device (such as counters, registers, or adders) are aligned with the logic resources of the chip. The exact process is device dependent. For example, FPGAs have look-up tables that perform logic operations. The mapping tool (part of the vendor's tool suite) collects the gates defined by the netlist into groups that will fit within the look-up tables.

Place and Route is the process of placing the logic blocks in the best spots on the chip to achieve efficient routing. Items that the place and route tool will look at include routing length (how far does a signal have to travel), track congestion (how many signals are coming into or out of an area), and path delays. While the process is usually performed automatically by the vendor-supplied tools, the designer can specify some parameters and constraints that the final layout has to meet, including:

- the initial placement of the cells
- a position for each physical connector
- a form factor

Programming the device

Once the design is successfully verified and found to meet timing and performance requirements, the final step is to actually program the device. At the completion of placement and routing, a binary programming file is created. It's used to configure the device. The process of programming is usually dependent on the type of memory used to store the device configuration and on the device type (e.g., FPGA or ASIC). So, let's look at some of the factors in device programming.

How Complex Electronic devices remember their configuration

User-programmable complex electronic devices are a combination of a logic device and a memory device. The memory is used to store the pattern that was given to the chip during programming. The primary ways this information is stored are:

- Fuses
- Antifuses
- SRAM
- flash memory
- (E)EPROM cells

A fuse is a special part of the programmable chip that is normally closed (connected) until an electrical current breaks that connection. Antifuses, unlike traditional fuses, are open until a voltage is applied to close (complete) the circuit path. Once programmed closed, the connection cannot be reprogrammed to open. Programmable logic that uses fuses or antifuses are “program once” chips. For operations on planet Earth, fuses and antifuses lag behind the more reprogrammable versions in versatility and market share. In applications where ionizing radiation is a concern (such as outer space or high altitude), antifuses are usually a better way to go.

SRAM, or static RAM, is a volatile type of memory. The contents of the memory are lost whenever the power is switched off. Static RAM differs from the dynamic RAM used in PCs in that memory refresh of the RAM is not required. SRAM-based programmable logic devices have to be programmed every time the chip is switched on. This is usually done automatically by another part of the system. SRAM FPGAs are susceptible to ionizing radiation, including the neutron radiation experienced at high altitudes.

An EPROM cell is a transistor that can be switched on by trapping an electric charge permanently on its gate electrode. This is done by an external programming device. The charge remains for many years and can only be removed by exposing the chip to strong ultraviolet light. EEPROM is electrically erasable PROM, which uses an electrical current rather than ultraviolet light to erase the programmed value. EPROMs have to be removed from their circuit boards to be erased and reprogrammed. EEPROMs can be erased and reprogrammed using special circuitry on the board.

Flash memory is non-volatile, which means that it retains its contents even when the power is switched off. It can be erased and reprogrammed as required. This makes it useful for programmable logic device memory. Flash-based devices combine the best of both worlds - maintaining configuration when not powered, but also allowing reprogramming when desired. Flash-based programmable devices are essentially immune to neutron radiation (generated when cosmic rays interact with the atmosphere) and are resistant to other high-energy particles.

Externally programmed devices

Complex electronics that use fuse, antifuse, or EPROM technology to configure the device have to be programmed in an external device, and cannot be programmed when placed on a circuit

board. EEPROM-based devices may also require external programming, or may be able to be programmed in-system, depending on the specifics of the device and the circuit.



To use an external programmer, the chip (CPLD or FPGA, or simple programmable logic device) is placed in the appropriate socket and attached to the programming device. The programmer is attached to a computer (or may have an internal microprocessor, for stand-alone devices), which will download the binary file into the device and then apply the necessary voltages to “burn” or program the chip.

In-system programming

Complex electronics that use SRAM, Flash, or (sometimes) EEPROM can be, and usually are, programmed in-situ on the circuit board. Many boards provide a JTAG interface that can be hooked up to a personal computer for download of the device configuration.

ASICs

Application Specific Integrated Circuits (ASICs) are *user designable*, not *user programmable* complex electronics. While the basic steps in designing ASICs is the same as for other complex electronics, there are some differences, driven by the fact that ASICs are manufactured (usually in large runs), and a problem with the resulting chip is very costly.

Here are some of the main differences between creating an ASIC and a programmable complex electronic device (e.g. FPGA):

Development Area	Differences
Vendor Selection	With FPGAs, you select which chip you will use. This is an off-the-shelf purchase, and the only question is whether the chip meets your needs and how good are the vendor-supplied tools. Since ASICs are manufactured, the vendor relationship is much more important. ASIC vendors will usually perform some of the implementation steps (such as place and route), as well as post-manufacture testing.
Careful Selection of Functionality	Because of the cost of failure with ASIC design, the selection of what functionality will be included in the ASIC is very important. While FPGA design, like software, can be changed later in the program, ASICs have a long lead time. So it's important to get everything right early in the process.
Simulation	With FPGA designs, the primary simulations are early in the design

Development Area	Differences
	<p>process, to verify the functionality of the design. With ASICs, simulations are mostly performed late in the design (at the gate level) to verify that all the last minute transformations and modifications don't cause an error. This difference affects what you want in a simulation environment. For ASICs, high performance (fast) simulation is essential. For FPGAs, the quality of the user environment and the speed in locating and fixing errors is more important.</p>
Design Size	<p>The size (in gates and/or I/O capacity) of ASICs is a somewhat continuous scale from small to very larger. FPGAs are "chunky" - the size varies in vendor-defined increments within a device family. ASICs have more flexibility in size, so that a small increase will not affect the final cost too much, whereas with an FPGA you might have to go to the next higher size (and more expensive) chip. In general, FPGA designers are more concerned with getting the design to the minimum size (or to fit within the target chip) than ASIC designers.</p>
Timing	<p>ASICs have a relatively smooth, continuous distribution of delays as routing distances vary. With FPGAs, delays move in large, discontinuous, and relatively unpredictable steps. This means the estimated timing performance can vary by 20-30% on a net-by-net and path-by-path basis between the various design tools.</p> <p>With ASICs, many timing problems can often be conquered by resizing buffers, small placement and routing changes, and cell swaps. These options are not available with FPGAs. Logic synthesis in FPGA can basically only replicate logic, rebalance trees, and restructure paths to resolve timing issues.</p>
Verification	<p>In the ASIC world, verification is a long and time consuming process. It will take up to 70% of the total development time and resources. The reason for this is risk avoidance - you don't want a design error to slip through and cause you to waste all the time and money spent on the ASIC. This makes verification, confirmation, and re-verification every design engineer's first priority. Verification of ASICs is a lot more rigorous than FPGA verification.</p> <p>Besides multiple simulations at various design phases, and design reviews with your best engineers, two additional verification activities may be performed:</p> <ul style="list-style-type: none"> • <i>Prototyping in FPGA.</i> Creating your ASIC design in an FPGA prototype results in discovery of bugs that may not have been identified during previous simulation. It also provides multiple platforms for software development in parallel to debugging. The FPGA prototype is available for

Development Area	Differences
	<p>the post-manufacture verification as an exerciser to validate the design.</p> <ul style="list-style-type: none"> • <i>Formal Methods.</i> Formal methods in the engineering world are those methods that use mathematical (formal) languages for writing specifications to prove that highest-level specifications are consistent with top-level objectives. They have an advantage over verbal prose and conventional simulators because they can be used to represent specifications that are provably consistent with objectives and higher level specifications.
Manufacturing	<p>Programmable complex electronics use off-the-shelf chips, whereas an ASIC design is submitted to a vendor for manufacturing. The manufacturing stage incurs significant expense. The vendor assumes responsibility for fabricating, probing and sorting wafers, then assembles and packages the chip per requirements. Once the chips are created (and pass the vendor tests), the designer has to complete verification of the final product - and hope that the design was correct! Any problems found will require the ASICs to be remanufactured, at a significant cost.</p>

SRAM-based FPGAs

Complex electronics that use SRAM will lose their memory once power is removed. Static RAM is volatile memory. So, these chips need additional resources in order to function. Since the configuration is lost whenever the power is removed, the FPGA configuration has to be placed in non-volatile memory, such as an EPROM, EEPROM, or flash memory. When the FPGA is powered on, it reads the configuration from the non-volatile memory and is ready to go.

Assurance Roles

At the implementation stage, assurance and safety engineers are usually not involved. A hardware assurance engineer might witness the programming (“burning”) of the device. Much of the implementation process is performed by automated tools, so if the tools were previously assessed, the results can be accepted without additional review. One area that the assurance disciplines can support at this time is verification the design and implementation is appropriate for the environment where the device will operate. NASA experts in radiation or other space-related effects can be consulted if there are any questions about the device design.

A software engineer taking this course should be able to witness the programming (“burning”) of a complex electronic device and to ask intelligent questions during the process.

4.7 Verification

As with software, verification activities do not wait until the complex electronic device is programmed and ready for test. Verification is a parallel set of activities to design and development. Various tasks are performed at each phase of the development.

This section of the course will answer the questions:

- What are the verification steps for complex electronics?
- How is verification for complex electronics similar to, or different from, software?
- Who performs the verifications?

Requirements

At the requirements phase, the system or sub-system level requirements are flowed down to the complex electronics. This flow down is primarily the responsibility of the systems engineer, though hopefully the design engineer for the complex electronics will be involved; to prevent requirements being imposed on the hardware that it cannot meet!

<i>Verification activity</i>	<i>Performed by</i>
Evaluate requirements for the complex electronics	Quality assurance, systems engineer
Safety assessment	System safety engineer
Requirements review (e.g. PDR)	All
Identification of applicable standards	Quality assurance, safety, design engineer
Formal methods	IV&V or knowledgeable practitioner

Quality assurance engineers should review the requirements for correctness, completeness, and consistency. High quality requirements are worth their weight in gold. Good requirements make everyone's life easier, because bad requirements are difficult to verify, are often interpreted differently by various people, and may not implement the functions that are desired. Finding out during testing that the device is missing important functionality, or is too slow, is something you really want to avoid.

For safety- or mission-critical devices, formal methods might be used as a verification tool. The requirements can be defined using a special language that allows mathematical proofs to be generated showing that the device will not violate certain properties. Formal methods can be applied at only the requirements level (to make sure you get those correct), or also be used to verify the design when it is generated. Most projects will not use formal methods.

Design Entry

During the design entry phase, the complex electronics functionality is defined in a hardware description language (HDL). The HDL code can be simulated in a test bench and its behavior can be observed. This is an important verification activity that is usually performed solely by the

design engineer. Quality assurance engineers may review the simulation plans (if they are produced) or results, and for critical devices they may witness some of the simulation runs.

<i>Verification activity</i>	<i>Performed by</i>
Evaluate design (HDL) against requirements	Quality assurance
Functional Simulation	Design engineer
Safety assessment	System safety engineer
Design review (e.g. CDR, peer review)	All
Static analysis of HDL code	IV&V, assurance engineer

Functional simulation involves emulating the functionality of a device to determine that it is working per the specification and that it will produce correct results. This type of simulation is good at finding errors or bugs in the design. Functional simulation is also used after the 'design synthesis' step where the gate-level design is simulated.

The HDL code should be reviewed by one or more engineers who can assess the design. A good reviewer has to understand the system within which the device will operate, know the HDL language being used, and be able to compare what the device is designed to do against its requirements. This means that not just anyone can adequately review the design. Lack of knowledge or experience will hamper the review, and often cause the designer to think it is a waste of time.

For very complex or safety-critical devices, Independent Verification and Validation (IV&V) may be called in to review the design. One tool they can use is static analysis software for the HDL code, which can look for problems or possible errors in the code. This tool is very similar to some static analysis tools for software that look for potential logic or coding errors.

Design Synthesis

During design synthesis, the higher level designs are optimally translated to a gate-level design, which can then be mapped to the logic blocks in a complex electronic device. It is during synthesis that timing and area constraints can be specified by the user.

<i>Verification activity</i>	<i>Performed by</i>
Functional Simulation of gate-level circuit	Design engineer
Design review (peer review)	All
Design evaluation	Quality assurance

Simulation is one of the primary ways that the design synthesis process is verified. In almost all projects, the design engineer is the one who generates the test bench, defines the simulation runs, and performs the simulations. Quality assurance is rarely involved, other than to perhaps verify that the simulations were performed. However, it is important to look at the design of the test bench and the simulation tests to make sure they are complete enough. This is the time to find

errors or mistaken assumptions - not when you are integrating your complex electronics with other areas of the system.

Implementation

Implementation is where the design meets the silicon - the higher level design is converted into a chip layout. The implementation process uses the tools supplied by the device (e.g. FPGA) vendor to match the functions that were defined in the design to the available blocks, gates, and other logic elements on the chip.

<i>Verification activity</i>	<i>Performed by</i>
Timing simulation	Design engineer
Static timing analysis	Design engineer
Device programming	Witnessed by Quality Assurance
Fault injection testing	Design engineer or quality assurance

Timing simulations are simply functional simulation with timing information. The timing information allows the designer to confirm that signals change in the correct timing relationship to each other. The timing information is entered in the hardware description language model file and then simulated. However, since there is a possibility of not being able to simulate all combination of inputs, a timing analysis tool can be used to evaluate a fully synchronous design.

Static timing analysis is a process that examines a synchronous design and determines its highest operating frequency. The analysis considers the path from every flip-flop in the design to every other flip-flop to which it is connected through the combinatorial logic. The analysis is usually performed by a software tool, which calculates the best-case and the worst-case delays through these paths (critical-paths). Any paths that violate the set-up or hold-timing requirements of the flip-flop are flagged for later adjustment to meet the design requirements.

Understanding how the complex electronics will operate when given invalid input is very important in verifying the devices. The real world is messy, and noisy signals or broken interfaced hardware are unfortunately common. Simulation is a great place to perform fault injection testing by inputting signals that are out of range, whose timing is not correct, that have ringing or other signal problems, or that are noisy. Encouraging this type of testing, and helping to identify the likely types of faults, is one way that quality assurance personnel can actively participate in the verification of complex electronics.

Testing

While simulation is used extensively in complex electronic design, testing the actual chip can sometimes be an eye-opening experience. Simulation involves assumptions and compromises that may not match with the real world. Testing the programmed chip - either independently or integrated onto a circuit board - is a necessary step in verifying your design.

<i>Verification activity</i>	<i>Performed by</i>
In-circuit functional and timing tests	Design engineer, may be witnessed by QA
Sub-system and system tests	All
Safety verification	All, but reviewed or witnessed by System Safety Engineer

In-circuit verification tests the functionality and timing of the design on the actual chip. Ideally, special test software running on a host computer will interface with the device under test through available test ports, such as the JTAG port. This process is similar to in-circuit emulators that run embedded software on the target processor, and provide breakpoints and tracing into the actual software instructions.

The more common form of in-circuit tests is to manually run the complex electronics as part of a higher-level assembly to show that it meets all the specified requirements. This sub-system or system level test will show functionality at a black-box level, but will not provide a window into the internal functioning of the device.

If the complex electronic device is safety-critical, there will be separate safety verifications, usually at the system level.

What should an assurance person look for when evaluating complex electronics?

Since you are taking this course, you are almost certainly not an expert in complex electronics. So what can you do, with limited experience, that will help improve the design process? At a minimum, you can ask questions of those producing or reviewing the design to help ensure that all important areas are considered. As a software assurance engineer, you may not be able to comment on the inner workings of the complex electronics, but you can certainly provide your “process assurance” viewpoint to the design and help make sure that defined processes are in place for configuration management, coding standards, and other areas. The following paragraphs provide some general guidance and questions to consider.

Programmatic Questions

- Is the design team experienced, or does it have at least one experienced member?
- Is there a design guideline document that defines design rules? How will the guideline help prevent a code-and-debug methodology as the design process?
- Has the team created a naming convention that provides information about objects and their timing information?

Is there an exception handling mechanism (possibly a hardware-software cooperative arrangement) for error conditions that may be detected?

- Is the design maintained in a version control or configuration management system? Is there a formal process for changes once the design is baselined?
- Has anyone looked at what standards may be applicable (Center, NASA, other)?

Design reviews

- Does the design meet the specification?
- Does the design pass a worst case analysis (timing)?
- Is the design partitioned into logical components?
- Does the designer provide enough background information that you can understand what the device is supposed to do?
- Is there anything in the design that conflicts with other sub-system or system components?
- Does the design interfaces (input and output signals) match the interfaces as specified by the other components?
- Review the special pins on each device (e.g., mode pin on FPGA, JTAG pins, no-connect pins) and verify that each is used properly.

Analyses

- Was a timing analysis performed with the following signals?
 - - Pulse width of each clock, asynchronous set, clear, and load input
 - Setup and hold time for all clocked inputs
 - Recovery time for set and clear
- Did the timing analysis also consider:
 - - Parallel clocking
 - Clock skew
 - Timing of analog circuitry
 - Minimum propagation delays
- Were the gate output drive capacities analyzed to show that none were exceeded?
- Were the interfaces to other parts analyzed for input logic level thresholds and maximum input transition times?
- If there is a state machine, was it analyzed for:
 - Unused states and lock-up
 - Simultaneous assertion of flip-flop sets and clears
 - Reset conditions
- Are resets of the correct assertion and release voltages, and is the pulse width correct?

5 Process Assurance

5.1 Process Assurance Overview

According to IEEE, quality assurance is defined as "a planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements." Quality assurance (QA) can be broken down into two main areas: product assurance and process assurance.

Product assurance involves making sure that the final product meets its specifications. This is usually done via thorough testing. Ideally, it also includes verifying that the requirements are correct, the design meets the requirements, and the implementation reflects the design.

Process assurance looks at the *process* used to create that final product. Was the development effort planned? Were the plans followed, or just put on the shelf and ignored? Does the development process meet any required standards? Are best practices used to develop the product? In process assurance, QA provides management with objective feedback regarding compliance to approved plans, procedures, standards, and analyses.

Process assurance activities are performed throughout the life cycle, including product conception, design, implementation, operation, and maintenance. Process assurance will detect, record, evaluate, approve, track and resolve deviations from approved plans and procedures. For each life cycle phase, process assurance makes sure that planning is performed, that the plan is followed, and that the products of each phase are correct and complete. Note that verifying the quality of the requirements, design, and verifications are usually considered *product* assurance. This course includes them in *process* assurance because they are often overlooked when evaluating complex electronics.

For a circuit board that is assembled, *product assurance* would include verifying that the correct parts are on the board, assessing the quality of the soldering, and testing the board functionality. *Process assurance* activities would include verifying that the drawing used during the board assembly was configuration controlled and the correct revision, that proper Electrostatic Discharge (ESD) requirements were followed, and that an assembly process was defined and followed.

5.1.1 Why do Process Assurance?

While some aspects of process assurance are performed in many engineering disciplines, process assurance is the cornerstone of software assurance. In some

industries, the main purpose of software quality assurance is to test the software prior to release. Within NASA, software assurance starts much earlier in the life cycle (with the requirements) and verifies the quality of all the products at each stage.

Why does software get this special treatment? Software differs from most “hardware” (mechanical or electrical) in several important ways:

- Software is complex and cannot be fully, 100% tested. It is not possible to test every possible path through the program, nor every combination of inputs. For more than a trivial program, such testing would take an astronomical amount of time.
- Software requirements are often fluid. Because software is so “easy” to change, many defects or problems with hardware systems are overcome by changing the software.
- Software itself is fluid. It is easy to add additional functionality just because it can be done - without sufficient thought as to the impact of that change on the entire system.

Using good practices to develop software increases the confidence in the quality of the software. Because you cannot fully test every combination of inputs and paths with software, you need a way to look at the whole development process and the test results and determine if the product is of sufficient quality. Process assurance is used to make sure those good practices are in place and that the project is following those practices.

Process assurance also looks at software throughout the life cycle and judges the quality of the process and the associated life cycle products. Software assurance engineers have a handle on the software requirements, design, and code volatility and can alert project management if too many changes are occurring. Because changes have an impact on other software or systems, software assurance engineers help identify and assess those impacts prior to the change being implemented. Process assurance is *proactive* in identifying and helping to correct potential problems before they become actual problems.

While some people may see process assurance as an unwanted but required activity, one of the main reasons to perform it is to embed quality throughout the life cycle. You don’t want to wait until the product is finalized before you have any idea if it is a quality product or not. Process assurance provides insight into the development processes (and thus some insight into the quality of the product) long before the product is completed. This focus on problem prevention through early detection allows corrections and changes to be made to the product or process when the cost of those changes is less than it would be later in the project.

5.1.2 Process Assurance for Complex Electronics

When software cannot be fully tested, process assurance (how the product is built) is used to increase confidence in the resulting program. The same philosophy can be

applied to complex programmable logic. In “Building a case for assurance from Process”², the author shows how process assurance can be used in the IT security world to make a case for claims about the software quality. This is the idea behind the Software-CMM and other process improvement initiatives. If you cannot verify by testing every possible combination of inputs, decisions, etc., then knowing that you built the software according to well-defined standards gives additional confidence.

Complex electronics are hardware, not software. But the design of these devices is complex enough that all combinations of inputs and timing cannot be fully tested. Complexity also increases the chance of design errors, unexpected interaction between elements of the design, and other “software-like” errors. Because of these concerns, complex electronics cannot be completely verified using traditional approaches.

Adding process assurance to the verification of complex electronics will increase the confidence that the final device was designed to the correct requirements, the design completely implements all requirements, and the final product meets all functional and quality specifications.

The Federal Aviation Administration (FAA) is taking a similar approach to complex electronics. The new standard DO-254, “Design Assurance Guidance for Airborne Electronic Hardware”, is basically process assurance for complex electronics. This standard requires:

- Planning for all life cycle phases, including selection of design methodology, integration of hardware design processes with supporting processes, and description of process assurance policies and procedures.
- Activities performed by engineers at each life cycle phase, including requirements capture, design creation, implementation, and acceptance testing.
- Verification and Validation throughout the life cycle.
- Configuration Management of designs and supporting information for complex electronics.
- Process assurance activities at each life cycle phase.

5.1.3 Tools of the Process Assurance Trade

Process assurance is implemented primarily through the following activities:

- Documentation Review
- Reviews, Walkthroughs, and Formal Inspections
- Audits
- Analyses

² “Building a case for assurance from Process”, K. Ferraiolo, L. Gallagher, V. Thompson; 21st National Information Systems Security Conference

The following paragraphs provide a quick overview of these processes. The next section of this course will go into more detail on which processes are appropriate for each phase of the life cycle, and what aspects of the complex electronics development they should be used for.

Documentation Review

Individual review of a document, design, or hardware description code is performed by the process assurance engineer. This type of review may or may not use a checklist (if one is available). The quality of the artifact is evaluated against best practices, and the results are fed back to the author of the artifact.

Reviews, Walkthroughs, and Formal Inspections

Formal inspection is an examination of the completed product of a particular stage of the development process (such as a design), typically employing checklists, expert inspectors, and a trained inspection moderator. The objective is to identify defects in the product. There are many techniques of doing inspections, but many follow the methods developed by Michael Fagan over 20 years ago.

Reviews are an alternative to formal inspections as a process assurance method. Informal design review methods are difficult to quantify since they are generally done at the discretion of the product author, do not follow a detailed process and are not reported at the project level. Informal review is a valuable alternative if the more effective formal inspection is not used.

Walkthroughs are meetings in which the author of the product acts as presenter to proceed through the material in a stepwise manner. The objective is often raising and/or resolving design or implementation issues. Walkthroughs tend to be informal and lacking in close procedural control.

Audits

A process assurance audit is performed to determine the level of adherence to the project plans and procedures. Evaluation of the sufficiency or effectiveness of the procedures and plans is occasionally part of an audit, though normally the evaluation is performed when the procedures and plans are first produced. This type of audit examines a sampling of records to determine if procedures are being followed correctly.

Records can include formal products (e.g., official design document), informal development information, log files, tool output files, and even emails. Configuration management and change control records are also often examined during a process assurance audit.

Analyses

Analyses are performed when required to evaluate an aspect of the system, a project artifact, or the impact of changes. For complex electronics, the specific analyses will depend on the device, the level of criticality, safety implications, life cycle phase, and other factors. An analysis can be as simple as a documented “expert review” or as complex as a computer simulation. The method used in performing the analysis needs to be documented, as well as the results.

5.2 Identifying Complex Electronics

Now that you know something about complex electronics, how do you recognize if your project is using it? How can you tell if the programmable devices are *simple* versus *complex*?

Simple versus Complex

Simple electronics includes off-the-shelf integrated circuits from simple logic devices up to microprocessors. While the software that runs on microprocessors is complex, the device itself can be considered simple because it is a) well tested by the manufacturer and b) not programmed at the hardware level by the end user.

The dividing line between simple and complex electronics is not well defined, and has not been officially determined by NASA. However, here are some guidelines to help you make the determination:

Simple	Off-the-shelf ICs Microprocessors PAL, GAL, PLA EPROM, EEPROM	These devices are either tested by the manufacturer or so simple that all inputs and outputs can be verified.
Gray area	CPLD	Depending on usage and size (gate count), CPLDs can be simple or complex.
Complex	FPGA System-on-Chip ASIC	These devices are too complex to be 100% tested.
Special Concerns	Distributed systems	Systems with one or more complex devices (or a complex electronic device and software) that jointly control a system or coordinate among themselves require assurance beyond the devices themselves. The interfaces and timing of communication are important to consider.

	Complex electronics as part of a off-the-shelf circuit board	Sometimes an FPGA or CPLD will be part of an off-the-shelf board. Since the design of the device is probably not available, you cannot perform any analysis or in-depth verification of the device. If the device is not used for safety purposes, it can probably be considered <i>simple</i> .
--	--	--

Programmable devices used as part of a safety system or hazard control should be assumed to be complex. To be considered simple, a very strong case should be made with sufficient analysis and documentation to justify the position.

How to determine if Complex Electronics is used in a project

If you, as a quality assurance engineer, are supporting a project, how can you determine if the project includes complex electronics? Here are some pointers:

- *Review project documentation.* Look at the system concept, any overviews or descriptions, and system and sub-system requirements and design documents.
- *Talk with the project system engineer and/or system safety.* The system engineer should be aware of any complex electronics. System safety should be aware of any complex electronics that are part of a hazard control or otherwise safety-related.
- *Talk with the project electrical engineer(s).* These are the people who will develop the devices.

What next?

If you find out that the project is using one or more complex electronic devices, what should you do next? The next step is to gather more information. Find out:

- what process is used for design of the devices?
- What tools are being used to design/develop/program the devices
- Is configuration management (CM) used? What about change management?
- How will the devices be verified?
- How will QA be involved in verifying and assuring these devices?
- What is the error handling philosophy in the design? Are there ways that external signal problems (invalid voltages, missing signals, etc.) can cause problems with the device?
- Are the devices safety-related? Do they acquire or process any signals used in safety decisions (e.g. temperatures, voltages)?
- Is the function of the device(s) mission-critical? Will failure seriously affect the ability of the system to carry out the mission?

If one or more of the devices are safety-related, share that information with the project system safety engineer. Safety-related complex electronics should be looked at by the system safety engineer in more depth.

If you see deficiencies in one or more areas (e.g. configuration management), you can research alternatives and make suggestions to the project manager or engineers on how to implement or improve the process. Configuration and change management are very important and often overlooked! You also want to guide the project away from a “program/debug/reprogram” paradigm, similar to undisciplined software development.

Be proactive. Get to know the device designers. Educate yourself on the devices, the tools used, and the design process. Do some web surfing for common errors with the devices, and make sure the designers have avoided them. Review the requirements for the device - are they clear and unambiguous? See if you can observe a simulation or two. Ask *intelligent* questions - ones that show that you are interested enough to have done some background work.

5.3 Process Assurance activities

Process assurance activities occur during all phases of a project life cycle. This section describes activities for each phase of the life cycle that are appropriate for complex electronics. Remember that there is currently no requirement for many of these activities, so implementing them on your project could require some negotiation. However, this information will help you apply your quality assurance expertise more thoroughly to complex electronics.

Quality assurance engineers need to possess sufficient domain knowledge to evaluate the completeness and correctness of complex electronics requirements and design. They must have the ability to determine whether the design has incorporated all requirements accurately. If you are not an electrical engineer, or have significant experience with complex electronics, you probably don't have that domain knowledge. For some process activities, you may wish to find an expert (either in the assurance arena or in engineering) to help you or to independently perform an analysis or evaluation. The most important aspect of assurance is evaluation by someone other than the designer, but not all evaluations have to be performed by the quality assurance engineer.

As you perform assurance activities on complex electronics, you should keep in mind some quality criteria. These criteria will help you judge the status of the product or process.

- *Correctness.* The extent to which a device fulfills its specifications.
- *Efficiency.* Use of resources; performance characteristics.
- *Flexibility.* Ease of making changes if required.
- *Security.* Protection of the device from unauthorized access.
- *Interoperability.* Effort required to couple the system to another system.
- *Maintainability.* Effort required to locate and fix a fault in the program within its operating environment.

- *Portability.* Effort required to transfer a device or design (program) from one environment to another.
- *Reliability.* Ability not to fail, including in off-nominal environments.
- *Testability.* Ease of testing the device to ensure that it is error-free and meets its specification.

5.3.1 Project Conception

The initial stage of a project or system is the time when many decisions are made that will affect the project months or years down the road. While the technical decisions are driven by the results of systems engineering trade-off studies, the assurance decisions are driven by a combination of:

- Requirements and Standards
 - What are the NASA, Center, and other quality assurance standards that the project must follow?
- Project management support.
 - The level of assurance is directly proportional to the amount of support that project management supplies. When quality assurance is perceived as a useful tool to help develop a functional system within the project constraints, QA is given adequate funds and personnel to do a thorough job. If the project manager deems QA an annoyance, then the ability of the QA engineer to implement an effective program is hampered.
- Effectiveness of the Assurance Organization
 - An assurance organization that has a track record of working with projects to develop tailored and effective assurance plans and processes will be more likely to gain project support in implementing new assurance activities. Conversely, an organization that does not have a good working relationship with projects will make it much more difficult for the assigned quality assurance engineer to persuade the project to consider any additional assurance activities for complex electronics.
- Knowledge and experience of the assurance professional
 - The individual providing QA has to be proactive in implementing quality assurance activities, especially for new areas such as complex electronics. If the QA engineer lacks knowledge and experience, the necessary assurance infrastructure may not be put in place.

Quality assurance is involved in project planning activities through:

1. Creation of a Quality Assurance Plan that outlines the work that will be performed by QA throughout the project life cycle.
2. Assessment of the project plans, including the management and development plans for electronics, for completeness, correctness, and other quality attributes.
3. Assurance that the project produces the required plans.

The plans a project will produce depends on the NASA and Center requirements and the project complexity and safety-criticality. The content of the plans often varies between projects, with one project combining several documents and others producing separate plans. Don't get hung up about which plan is which, but review the project plans for how they will address complex electronics. If they don't address the issues at all, try to encourage the project manager or the design engineer to at least informally document the information.

Here are some areas the project plans should address regarding complex electronics:

- What life cycle will be used to develop the complex electronics? How will the CE life cycle interface with the project life cycle? In describing the life cycle, does the document discuss transition criteria between phases, and how to return to previous phases if problems are found?
- Are there standards that apply to the complex electronics? NASA currently has no defined standard, though DO-254 is pending acceptance as a preferred standard.
- What is the hardware design process?
 - What activities will be performed as part of the process?
 - How will the hardware design process work with supporting processes, such as verification and assurance?
 - Is the design method(s) for complex electronics defined and described?
 - What design environment (e.g. tools) will be used? What is the rationale for the selection?
- If deviation from established plans becomes necessary, what is the process for doing this? For example, how will changes be approved by all interested parties?
- How will the design for complex electronics and any associated data be included in the configuration management system?
 - What process is in place to review and approve any revisions to the design?
- Are the plans completed before the life cycle phase they will be used in? Plans for configuration management should be finalized before development starts, for example.

The Quality Assurance Plan should include activities for reviewing the requirements and design, witnessing or performing testing, and other product verification steps. The plan should also include formal or informal audits to verify that the project is following the plans they defined. If a project plan is just gathering dust, it's important to look for the reason. Maybe the document is too high level. Maybe things have changed enough that the document is out of phase with project reality. Whatever the reason, try to work with the project to fix the document problems so that the plans are useful and relevant.

Risk management is an important tool that projects can use in reducing the probability or impact of risks. Complex electronics has some similarities to software, including the fluidity of the requirements, interface problems with other elements of the system, integration issues (often a result of the interface problems), and the need to create a

complex “program” within a defined period of time. These types of issues are ideal for risk management mitigation!

5.3.2 Requirements

During the requirements development process, system requirements are allocated to various sub-systems and parts, including complex electronics. These requirements need to be documented (in a separate specification for complex electronics or as part of another requirements specification document).

The process assurance activity is to review the requirements for the complex electronic devices and verify that they:

- Include all requirements that are appropriate for the complex electronics (i.e. that the allocation was complete)
- Identify any requirements that are safety-related
- Identify design constraints for the complex electronics
- Are clear, concise, and verifiable
- Are traceable to a higher level document or are noted as derived requirements

It is important that the requirements are as clear as possible, because many problems found later in system design can be traced back to ambiguous or incorrect requirements. Requirements for complex electronics should also be more than just a cut-and-paste from the system requirements specification. They should be decomposed to the appropriate level of detail, and provide enough information that a designer can go off and create the device.

Activities for the verification of requirements for complex electronics must also be specified in the verification plan. If a verification method cannot be determined, that indicates that the requirement is flawed and needs to be fixed.

5.3.3 Design Entry

During the design entry phase, the complex electronics functionality and structure are defined in a hardware description language (HDL). The HDL “program” is actually a model of the complex device, and can be “run” (simulated) and tested. This phase is when any problems with the requirements should be identified and the high-level functionality should be verified.

Prior to the start of the design, several process assurance activities should be performed:

- **Tools.** Review selected tools for applicability to the design process. Check the tool vendor web-site and other sources for known tool defects or operational workarounds.

- **Design Process.** Make sure a disciplined design process is in place, and the design engineer is willing to follow it. Negotiate as necessary.
- **Configuration Management.** Make sure the HDL code and other design information is configuration managed. The level of formality depends on status of design (e.g. informal version control prior to baseline, formal change control after baselining).
- **Design and Coding Standards.** Ensure that the design team is using a design and coding standard. This standard will define the basic design philosophy and specify aspects of the HDL program structure. Even if only one engineer is designing the device, a standard 1) helps ensure that the HDL program is understandable by others (and the design engineer, six months down the road) and 2) provides a way to capture and incorporate best practices in the design process.

A design and coding standard should include:

- Specific HDL coding features and methods that either should be used, or should not be used.
- Design “best practices”, either as a guideline or as requirements.
- Naming conventions for modules, inputs, outputs, etc.
- Commenting rules that define what types of information to include in comments. One example would be to define a module header that includes comments on the module’s purpose and structure.
- Readability rules may be covered under naming and commenting conventions. But the standard should help guide the designer into creating HDL code that is readable by others.
- Modularization guidelines provide information on how to decompose the high level design into individual modules.

Assessment of the HDL design can be performed in parallel with the design effort, with intermediate design elements being reviewed, if the project criticality warrants it. Otherwise, the review is normally performed after at least a fairly stable design (if not baselined) is created.

Process assurance activities post-HDL-design include:

- Ensure that the design is reviewed by someone knowledgeable enough to make an expert assessment. This can be another engineer, a quality assurance engineer, or even an outside expert. Another set of eyes will help spot problem areas of the design. This review could be part of a Formal Inspection or other peer review.
- Review the design (behavioral and structural specification in HDL) against the requirements. Are all requirements correctly and completely implemented?
- Trace the requirements into the design elements. The rigor of this tracing should be determined by the safety- and mission-criticality of the device.
- Identify any derived requirements that emerge from the design process. Make sure the rationale for these requirements is captured.
- Review the design against the design and coding standard.

- Assess the design for unused functions.
- Assess the use of special pins on each device (e.g. mode pin on FPGA, JTAG pins, no-connect pins) and verify that each is used properly.
- Identify constraints (design, installation, operation) that could affect safety if not followed.
- Assess the simulations that were performed. Did they cover all the required functionality? Were all modules exercised?
- Verify that the processes defined in the project plans were adhered to.
- Assure that any design trade-offs done for speed, size, etc are documented.

5.3.4 Design Synthesis

During design synthesis, the higher level designs are translated to a gate-level design, which can then be mapped to the logic blocks in a complex electronic device. This step also optimizes the design to make the most efficient use of the target device. It is during synthesis that timing and area constraints can be specified by the user.

Process assurance activities at this phase are:

- Verify that the design process, as defined in the project plans, was followed.
- Verify that the tools specified in the previous phase are the ones that are being used.
- Verify that the configuration management system is being used as defined in the project plans.

Additional assurance activities require someone with expertise in complex electronics. They can be performed by the quality assurance engineer or by an engineer independent of the project.

- Evaluate the test bench that was created by the design engineer for adequate testing capability of the device design.
- Review the constraints specified by the design engineer as input to the synthesis process for reasonableness.
- Assess the simulations that were performed after design synthesis is completed. Did the addition of timing information affect the outcomes of the simulations? Did the simulations look at worst-case timing, including on incoming signals?

5.3.5 Implementation

During the implementation phase, the higher level design is converted into a chip layout. The implementation process uses the tools supplied by the device vendor to match the functions that were defined in the design to the available blocks, gates, and other logic elements on the chip.

Much of the implementation process is performed by automated tools, so the assurance and safety engineers are usually not involved in any depth. Some process assurance activities at this phase are:

- Verify that the design process, as defined in the project plans, was followed.
- Verify that the tools specified in the project plans are the ones that are being used. Note any discrepancies and the rationale for using a different tool.
- Verify that the configuration management system is being used as defined in the project plans.
- Ensure that timing simulations or static timing analyses were performed.
- Verify that the simulations performed included out-of-range inputs, inputs that arrived in an incorrect order, and other “real world” problems that can be anticipated.
- Verify that the device is programmed according to a defined process and that it is witnessed by appropriate personnel (usually quality assurance).
- Were the interfaces to other parts analyzed for input logic level thresholds and maximum input transition times?
- If there is a state machine, was it analyzed for:
 - Unused states and lock-up
 - Simultaneous assertion of flip-flop sets and clears
 - Reset conditions

5.3.6 Testing

Once the device is programmed, it should be tested with other components. Initial testing may occur in a breadboard system, with final (acceptance) testing occurring in the real hardware system. This in-circuit verification tests the functionality and timing of the design on the actual chip.

The more common form of in-circuit tests is to manually run the complex electronics as part of a higher-level assembly to show that it meets all the specified requirements. This sub-system or system level tests will show functionality at a black-box level, but will not provide a window into the internal functioning of the device.

Process assurance activities for this phase include:

- Verify the defined processes are in place and are being followed correctly.
- Verify that the testing strategy has been documented in a plan and/or procedure, and that testing occurs according to the plan.
- Verify that the planned tests will completely verify the requirements in all reasonably expected situations. This includes verifying the functionality and performance in nominal situations and when other parts of the system have errors. How gracefully does the device handle errors it may encounter? How gracefully can it handle any internal faults?
- Verify that the planned tests will exercise all modules or other division in the device. Not every level of testing has to exercise all modules, but each module should be tested at some level (device, circuit board, sub-system, or system).

- Review the test plans and procedures to identify any areas where testing is weak. You are looking for modules that are only minimally tested, requirements that are only verified under some circumstances, and other areas where additional testing may be helpful.
- Witness tests (as agreed to in the project plans) and document any anomalies and problems.
- Review the test results to verify that no unnoticed anomalies occurred. Sometimes during testing many events are occurring and an anomaly unrelated to the aspect of the particular test may be missed.

5.3.7 Operations and Maintenance

Once the system is operational, the role of process assurance is not over. While the original project assurance engineer may have moved on to another project, some assurance engineer should be maintaining a minimal role with the system.

Process assurance activities during operations and maintenance include:

- Review operational and maintenance procedures for inclusion of any workarounds or other information that was discovered during development and testing.
- Support any failure review boards or help assess any problems that are identified during operations.
- If the complex electronic device is to be reprogrammed, assess the impact of the changes on the device, the system, and operational procedures.

5.3.8 Supporting Processes

Configuration Management (CM)

Configuration Management is, unfortunately, not often used for complex electronics design artifacts. The final design is usually saved, but the intermediate development artifacts are under the control of the designer. While formal configuration management might not be necessary until the design is finalized (baselined), some form of informal control (e.g., use of a version management system) is recommended. Being able to revert to a previous version of the design is useful when problems are discovered during development. Being able to recreate versions of the design might also be useful to help narrow down when a problem was introduced.

Once the design is baselined, formal configuration management should be applied to the design. CM includes change control. This means that a process is in place for any changes to be approved prior to the changes being implemented. Often a Configuration Control Board or an engineering board is used to review and approve (or disallow) the changes. Change control assures that:

- Changes to one part of the system do not adversely affect other parts of the system.
- The configuration of the device is always known (i.e., there are not unauthorized changes).
- Everyone who may be affected by the change has a chance to evaluate the change for impacts to their area of concern.

Reliability

Most reliability studies look at the hardware failure rates for the devices in a system. While failure of the actual device (e.g., FPGA) can be known, the failures related to design errors or unexpected interactions within the FPGA, once it is programmed, are not easy to determine. Most reliability evaluations ignore software for this very reason.

While there is currently no good way to assess the reliability of a complex electronic design, the fact that there may be design errors should be considered by the reliability engineer. At a minimum, the confidence in the resulting numbers (mean-time-to-failure, system reliability) is lowered.

Maintenance and Maintainability

If the device will potentially need to be maintained (including reprogramming updates), this issue needs to be considered early in the design of the complex electronics and its supporting circuitry. Some areas to consider are:

- Will the device architecture allow for the types of enhancements that can be foreseen?
- Does the design specification provide the information that an engineer would need to understand how the product works?
- Is the HDL code readable?
- Are comments liberal and informative?
- Is the necessary physical infrastructure in place to allow reprogramming?
- Is access to the reprogramming port, if one is used, available when the system is installed?

6 Future Trends

6.1 *Changes in CE design and verification*

Technology never stands still. Within the realm of complex electronics, devices such as System-on-Chip, FPGAs with embedded microprocessors, and reconfigurable computing all strain the traditional hardware-oriented design and verification approaches. Increasing complexity in designs also make it harder for the designer to conceptualize the design. Several new methods in design and verification of complex electronics will hopefully help improve verification of these devices.

Hardware/software codesign and coverification

Since complex electronics is increasingly being combined with software, codesign (and subsequently, coverification) of the hardware and software is a good idea. Hardware/software codesign is the cooperative design of hardware and software, within a single chip or within a system. One of the goals of codesign is to shorten the time-to-completion while reducing the design effort and costs of the designed products.

In hardware-software codesign, designers consider trade-off in the way hardware and software components of a system work together to exhibit a specified behavior, given a set of performance goals and technology. This trade-off between hardware and software illustrates the optimization aspect of the codesign problem. Codesign is an interdisciplinary activity, bringing concepts and ideas from different disciplines together (e.g., system-level modeling, hardware design and software design).

Current development methods for designing embedded systems and complex electronics require specification and design of the hardware and software as separate entities. A specification, often incomplete, is developed and sent to the hardware and software engineers. The hardware-software partition is decided early on in the project life cycle and is adhered to as much as is possible, because any changes in this partition may necessitate extensive redesign. Designers often strive to make everything fit in software, and off-load only some parts of the design to hardware to meet timing constraints.

The codesign process starts with specifying the system behavior at the system level. After this, the system specification is divided into a set of smaller pieces, so-called granules (e.g., basic blocks). Trade-off studies are performed to determine the most effective way to partition the functionality into hardware and software. The granules are mapped to hardware and software, resulting in sets of granules implemented on hardware (hardware parts) or software (software parts). Once the mapping is done, the implementation-independent system specification is decomposed into hardware and software specifications.

Hardware is synthesized from the given specification; the software specification is compiled for the chosen processor. The result of this co-synthesis phase is a set of complex electronics and a

set of assembler programs for the processors. In a final co-simulation step, the complex electronics are simulated together with the processors executing their generated assembler programs. The results are iterated until a sufficient system implementation has been found.

The coverification problem in system-level design includes different methods to detect errors at different abstraction levels. Coverification methods include formal verification, simulation or emulation. Formal verification formally proves either the equivalence of different design representations or specific properties (e.g., the absence of dead-lock conditions, of the system specification).

Simulation validates the functional correctness for a set of input stimuli. In most cases, only a small set of all combinations of input stimuli can be simulated. For this reason, simulation only ensures the correct behavior with a certain probability. Simulation can be applied during different design steps including the co-simulation step after co-synthesis.

To speed up the simulation time for simulating a partitioned hardware/software system, emulation is used. Emulation systems couple the complex electronics (either the programmable devices or, for ASICs, a programmable equivalent) with processors on a board. Therefore, emulators are the closest representation of real prototypes that is possible.

System modeling

Hardware description languages (HDLs) allow you to model the system at various levels of abstraction. However, they are still fairly “low level” abstractions, representing the hardware aspects of the design. Several new modeling languages, and extensions to existing languages, allow higher-level modeling of the system.

The purpose behind higher level modeling is to:

- Keep the design at a level of abstraction that human minds can grasp. Complex designs make it difficult for a human to understand both the device and how it interacts with its environment.
- Verify the design at a high level, then allow tools to generate the low-level design.
- Model the complex electronics as part of a larger system that includes software and possibly biological constructs.

Researchers and industry are developing system modeling languages or language extensions for use in complex systems. There are two parts to a system design language: the ability to express ideas in a natural language and a component that can translate the functions into working architectural components. Here are two areas of language development that are being actively pursued:

- *Using C or C++* to model the system. One product, SystemC, provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++. It can be used from initial concept to implementation in hardware

and software. SystemC provides an interoperable modeling platform which enables the development and exchange of very fast system-level C++ models. It also provides a stable platform for development of system-level tools.

- *SystemVerilog* is a proposed new standard, enhancing Verilog so that it provides built-in support for a wide range of modern design and verification methodologies. SystemVerilog is an extension to the Verilog language, which enables the modeling and verification of systems at a high level of abstraction. It adds a significant set of language enhancements on top of the Verilog 2001 standard, including features for high-level, abstract system modeling, testbench automation, and the integration of Verilog with the C programming language.
- *MATLAB and Simulink* can be used to model systems. MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numerical computation. Simulink is a platform for multi-domain simulation and model-based design of dynamic systems. It provides an interactive graphical environment and a customizable set of block libraries that let you accurately design, simulate, implement, and test time-varying systems, including control systems, signal processing, and communications.

6.2 NASA assurance changes

Currently, within NASA, complex electronics are treated as hardware devices only. The design of complex electronics may be reviewed by quality assurance engineers, the assembly into a board or system is witnessed and/or verified by quality assurance, and the final resulting electronic device is tested. However, the complex nature of these devices requires additional assurance effort beyond that given to an off-the-shelf component. Hardware quality assurance personnel may not be fully cognizant of the functions, potential problems, and issues with these devices.

At NASA Headquarters, this assurance problem is being discussed and debated. What types of assurance activities should be applied to complex electronics? Who should be involved in the assurance of these devices? What competencies are necessary to provide adequate assurance of complex electronic devices?

The Federal Aviation Administration (FAA) faced similar concerns several years ago. They discovered that software functions were being implemented in FPGAs, to avoid having to follow the FAA software assurance standard (DO-178B). The FAA struggled with the problem, and finally came up with a standard for Complex Electronic Hardware (CEH) that is similar to the software standard. CEH includes the complex electronic devices discussed in this course, and some additional devices. The resulting standard, DO-254, “Design Assurance Guidance for Airborne Electronic Hardware”, provides guidelines on the use of process assurance for complex electronic hardware.

One direction that NASA is looking at is similar to the FAA approach of implementing process assurance. Software is a very complex entity that cannot be fully tested. In the software world, process assurance (evaluating how the product is built) is used to increase confidence in the

resulting program. The same philosophy can be applied to complex electronics. If you cannot verify by testing every possible combination of inputs, decisions, etc., then knowing that you built the device according to well-defined standards gives additional confidence in its quality.

Process assurance will look at all life cycle stages of complex electronics development, from requirements to operations. “Process assurance” is very similar to the process part of software assurance, where we verify that the software development process was planned and the plan was followed, where requirements are reviewed and evaluated, the software design is evaluated against the requirements, code may be inspected or reviewed, and finally the resulting software is verified against the requirements. For hardware, the same types of activities are performed.

As a software assurance engineer, you are probably wondering what your role may be in the future. Since software assurance encompasses process assurance, software assurance engineers are well versed in the ideas and concepts. What is lacking is the knowledge to assess complex electronics. In order to effectively carry out assurance duties for complex electronic hardware, a software assurance engineer must understand 1) the hardware itself, 2) the process and language used to design the device and 3) how and when to apply software-style assurance techniques to the device.

This guidebook is one step in educating NASA software and quality assurance and system safety engineers on the design and verification of complex electronics. By itself, this guidebook *will not* make you an expert able to perform assurance of the devices. However, you have hopefully learned enough to be able to speak intelligently with hardware design engineers and to think about the various issues that need to be addressed. If nothing else, this guidebook has raised your awareness of complex electronics, and you can more intelligently apply quality product and process assurance to these devices.

Appendix A

A.1 Acronyms

A/D	Analog to Digital
ABEL	Advanced Boolean Equation Language
ADC	Analog to Digital Converter
ASIC	Application Specific Integrated Circuit
CE	Complex Electronics
CEH	Complex Electronic Hardware
CM	Configuration Management
CMM	Capability Maturity Model
CPLD	Complex Programmable Logic Device
CUPL	Cornell University Programming Language
D/A	Digital to Analog
DSP	Digital Signal Processor
EEPLD	Electrically Erasable Programmable Logic Device
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPLD	Erasable Programmable Logic Device
EPROM	Erasable Programmable Read-Only Memory
FAA	Federal Aviation Administration
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GAL	Generic Array Logic
GPS	Global Positioning System
HDL	Hardware description language
I/O	Input/Output
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
ISS	International Space Station
JHDL	Java Hardware Description Language
JTAG	Joint Test Action Group
IT	Information Technology
MAPLD	Military-Aerospace Programmable Logic Devices (a yearly conference)
PAL	Programmable Array Logic
PDA	Personal Digital Assistant
PL	Programmable Logic
PLA	Programmable Logic Array
PLC	Programmable Logic Controller
PLD	Programmable Logic Device
QA	Quality Assurance
RAM	Random Access Memory
SA	Software Assurance
SEI	Software Engineering Institute
SoaC	System-on-a-Chip
SoC	System-on-Chip
SRAM	Static Random Access Memory
VHDL	Very High Speed Integrated Circuit Hardware Description Language

A.2 Glossary

Antifuse	A two-terminal device that is normally a high resistive element and is programmed to a low impedance.
Architecture	The common logic structure of a family of programmable integrated circuits. The same architecture may be realized in different manufacturing processes.
ASIC (Application Specific Integrated Circuit):	IC product customized for a single application.
Asynchronous	A signal whose data is acknowledged or acted upon immediately, irrespective of any clock signal.
Boundary scan	Boundary scan is a methodology allowing complete controllability and observability of the boundary pins of a JTAG-compatible device via software control. This capability enables in-circuit testing without the need of in-circuit test equipment.
Cell Library	The collective name for the set of logic functions defined by the manufacturer of an Application-Specific Integrated Circuit (ASIC). The designer decides which types of cells should be realized and connected together to make the device perform its desired function.
Chip	Another name for an integrated circuit.
Codec	Short for compressor/decompressor, a codec is any technology for compressing and decompressing data. Codecs can be implemented in software, hardware, or a combination of both. Some popular codecs for computer video include MPEG, Indeo and Cinepak.
Combinatorial	A digital function whose output value is directly related to the current combination of values on its inputs. Also known as combinational.
Comparator (digital)	A logic function that compares two binary values, and outputs the results in terms of binary signals representing less-than and/or equal-to and/or greater-than.
Configurable/Complex Logic Block (CLB)	The array of multi-input and multi-output logic cells to be programmed. CLB is a configurable logic block that consists mainly of Look-up Tables (LUTs) and flip flops.
Cores	In the semiconductor design industry, refers to predefined functions such as processors or bus interfaces that are typically licensed from the software developer. Cores can be implemented directly in silicon, either in fixed logic or programmable logic devices, and saves chip designers time during product development. Synonymous with Intellectual Property.
CPLD (Complex Programmable Logic	Programmable logic devices characterized by an architecture offering high speed, predictable timing and simple software.

Device)	
Die	An unpackaged integrated circuit. In this case, the plural of die is also die.
Digital Signal	A digital signal is one whose key characteristic (e.g., voltage or current) fall into discrete ranges of values. Most digital systems utilize two voltage levels (low and high values).
Digital Signal Processor (DSP)	A specialized CPU used for digital signal processing of signals such as sound, video, and other analog signals which have been converted to digital form. Some uses of DSP are to decode modulated signals from modems, to process sound, video, and images in various ways, and to understand data from sonar, radar, and seismological readings.
EEPROM (Electrically-Erasable Programmable Read-Only Memory)	A memory device whose contents can be electrically programmed by the designer. Additionally, the contents can be electrically erased allowing the device to be reprogrammed.
Electro-Static Discharge (ESD)	The term electro-static discharge refers to a charged person, or object, discharging static electricity. Although the current associated with such a static charge is low, the electric potential can be in the millions of volts and can severely damage electronic components.
EPROM (Erasable Programmable Read-Only Memory)	A memory device whose contents can be electrically programmed by the designer. Additionally, the contents can be erased by exposing the die to ultraviolet light through a quartz window mounted in the top of the component's package.
Falling-Edge	A transition from a logic 1 to a logic 0. Also known as a <i>negative edge</i> .
Firmware	Software programs, or sequences of instructions, that are hard-coded into non-volatile memory devices.
FIFO First-in first-out	A data structure or hardware buffer where items come out in the same order they came in.
Flash memory	Non-volatile storage device similar to EEPROM, but where erasing can only be done in blocks or the entire chip.
Flip-flop	A digital logic circuit that can be switched back and forth between two states.
FPGA (Field Programmable Gate Array)	High density PLD containing small logic cells interconnected through a distributed array of programmable switches. This type of architecture produces statistically varying results in performance and functional capacity, but offers high register counts. Programmability typically is via volatile SRAM or one-time-programmable antifuses.
Fuse	A two-terminal device that is normally a low resistive element and is programmed or "blown" resulting in an open or high impedance.
Gate	In electronic circuitry, a pathway that may be open or closed, depending on the source of the input, the strength of a signal,

	or the conductivity of chemicals used in semiconductors. Logic gates are programmed to correspond to related "if-then" statements. The state of an open or closed gate is analogous to the binary state of a 0 or a 1. The application of this analogy allows computing machinery with millions of gates to respond conditionally and to perform logical functions.
Gate Array	IC that is customized by interconnecting an array of logic elements. Customization is performed by the manufacturer and typically involves non-recurring engineering (NRE) costs and several design iterations.
Glue	Generic term for any interface logic or protocol that connects two component blocks. Hardware designers call anything used to connect large VLSIs or circuit blocks "glue logic."
Hardware Description Language (HDL)	A kind of language used for the conceptual design of integrated circuits. Examples are VHDL and Verilog.
IC (Integrated Circuit)	A device in which components such as resistors, capacitors, diodes, and transistors are formed on the surface of a single piece of semiconductor.
In-Circuit Reconfigurable (ICR)	An SRAM-based, or similar component which can be dynamically reprogrammed on-the-fly while remaining resident in the system.
In-System Programmable (ISP)	An EEPROM-based, FLASH-based, or similar component which can be reprogrammed while remaining resident on the circuit board.
JEDEC (Joint Electronic Device Engineering Council)	A council which creates, approves, arbitrates, and oversees industry standards for electronic devices. In programmable logic, the term JEDEC refers to a textual file containing information used to program a device. The file format is a JEDEC approved standard and is commonly referred to as a JEDEC file.
JHDL	JHDL is a method of describing (programmatically, in JAVA) the components and connections in a digital logic circuit. More specifically, JHDL provides object classes used to build up circuit structure.
JTAG	Joint Test Action Group (JTAG, or "IEEE Standard 1149.1"). A standard specifying how to control and monitor the pins of compliant devices on a printed circuit board.
Logic	One of the three major classes of ICs in most digital electronic systems: microprocessors, memory, and logic. Logic is used for data manipulation and control functions that require higher speed than a microprocessor can provide.
Logic Function	A mathematical function that performs a digital operation on digital data and returns a digital value.
Logic Gate	The physical implementation of a logic function.
Logic Synthesis	A process in which a program is used to optimize the logic used to implement a design.

Look-Up Table (LUT)	An array or matrix of values that contains data that is searched. An alternative implementation of a CLB; the multiple inputs generate the complex outputs.
Macrocell	A macrocell on most modern CPLDs contains a sum-of-products combinatorial logic function and an optional flip-flop. The combinatorial logic function typically supports four to sixteen product terms with wide fan in. Thus, a macrocell may have many inputs, but the logic function complexity is limited. On the other hand, most FPGA logic blocks have unlimited complexity, but the logic function only has four inputs.
Netlist	A list of names of symbols or parts and their connection points, which are logically connected in each net of a circuit. A file listing parameters extracted from a circuit schematic.
Noise	The miscellaneous rubbish that gets added to a signal on its journey through a circuit. Noise can be caused by capacitive or inductive coupling, or from externally generated interference.
Non-volatile	The memory elements keep their contents when power is removed from the device.
Onboard	Contained on the device or on the board.
One Time Programmable	This device can be programmed only once; it's contents can not be changed. While typically these devices are fuse or antifuse based, they can also be low-cost EPROM devices. In this case, typically used for production devices, an inexpensive package is used without a window.
Partial Reprogrammability	The ability to leave the internal logic in place and change just one part of the FPGA.
Pinout	A diagram that indicates how wires are terminated to pins in a connector. A list that assigns device functions to specific pins.
Place and Route	Using backend implementation software tools, the process of connecting various memory elements in an FPGA to create a custom logic circuit.
Programmable Logic	A logic element whose function is not restricted to a particular function. It may be <i>programmed</i> at different points of the life cycle. At the earliest, it is programmed by the semiconductor vendor (standard cell, gate array), by the designer prior to assembly, or by the user, in circuit.
Programmable Logic Controller	A control device, usually used in industrial control applications, that employs the hardware architecture of a computer and relay ladder diagram language. Inputs to PLC's can originate from many sources including sensors and the outputs of other logic devices. Also called "programmable controller".
Reconfigurable Computing	A methodology of using programmable logic devices in a system design such that the hardware-based logic can be changed to perform various tasks. Benefits includes the use of fewer components, less power, and the flexibility that bring

	about. Also allows networked equipment in the field to be upgraded or repaired remotely.
Reprogrammable	These devices can have their configuration loaded more than once. SRAM-based devices may be reloaded without restriction. Many other forms of reprogrammable elements have restrictions on the number of write cycles, although they are high enough not to be of practical concern for most applications.
Rising-Edge	A transition from a logic 0 to a logic 1. Also known as a <i>positive edge</i> .
RTL Register Transfer Level	Register transfer level description, also called register transfer logic, is a description of a digital electronic circuit in terms of data flow between registers, which store information between clock cycles in a digital circuit. RTL description specifies what and where this information is stored and how it is passed through the circuit during its operation.
Sensor	A transducer that detects a physical quantity and converts it into a form suitable for processing. For example, a microphone is a sensor which detects sound and converts it into a corresponding voltage or current.
SRAM Static Random Access Memory	A type of memory that is faster and more reliable than the more common DRAM (dynamic RAM). The term static is derived from the fact that it doesn't need to be refreshed like dynamic RAM, but it loses its memory if it's powered off.
Standard Cell	This device differs from the gate array since each cell may be different and optimized for each "standard" function. There are no standard layers to the device and each layer of the chip is a unique design.
State Machine	The actual implementation (in hardware or software) of a function that can be considered to consist of a set of states through which it sequences.
Switch	A device for making or breaking an electric circuit, or for selecting between multiple circuits.
Synchronous	(1) A signal whose data is not acknowledged or acted upon until the next active edge of a clock signal. (2)A system whose operation is synchronized by a clock signal.
Trace	A line or "wire" of conductive material – such as copper, silver, or gold – on the surface of or sandwiched inside a PCB, printed circuit board. These traces are often called individually a run. Traces carry an electronic signal or other forms of electron flow from one point to another.
Truth Table	A convenient way to represent the operation of a digital circuit as columns of input values and their corresponding output responses.
Verilog	A Hardware Description Language for electronic design and gate-level simulation.

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language	A Hardware Description Language for electronic design and gate-level simulation.
Via	Feed-through. A plated through-hole in a printed circuit board used to route a trace vertically in the board, that is, from one layer to another.
Volatile	The memory elements lose their contents when power is removed from the device. SRAM-based devices are volatile and require another device to store their configuration program.

A.3 Links

NASA-related Links

URL	Description
http://klabs.org/	NASA Digital Engineering Institute
http://klabs.org/richcontent/Tutorial/tutorial.htm	Tutorials
http://ehw.jpl.nasa.gov/	JPL Evolvable Hardware laboratory
http://nepp.nasa.gov/index.cfm	NASA Electronic Parts and Packaging Program

Other Links

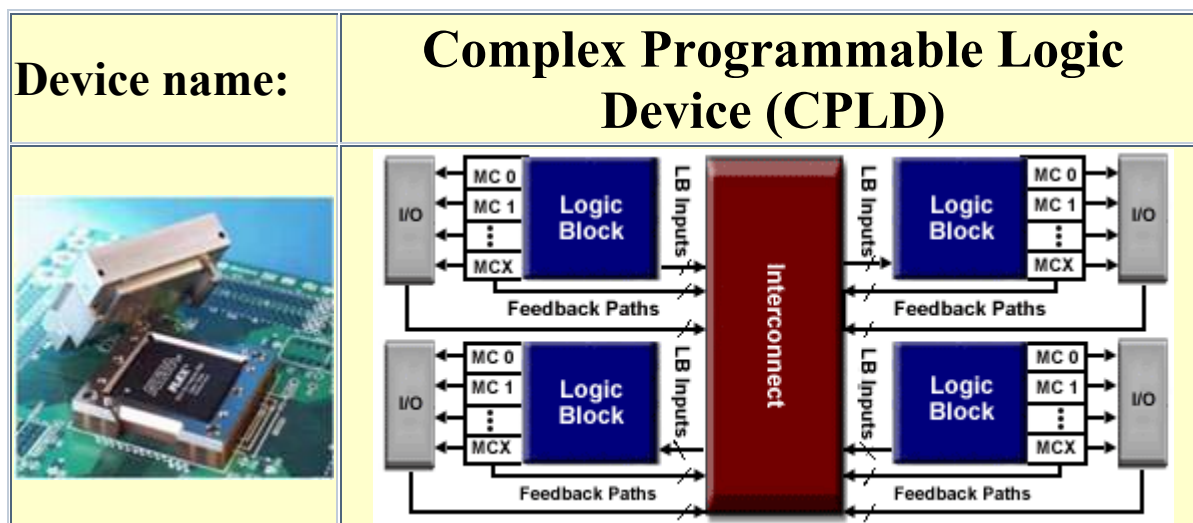
URL	Description	FPGA	Reconfig. Comput.	VHDL	Verilog	Other
http://www.icd.com.au/vhdl.html	Tutorials	x	x	x		
http://www.epanorama.net/links/fpga.html	Information	x		x		
http://www.cotsjournalonline.com/home/article.php?id=100043	Article	x	x			
http://www.vol.webnexus.com/	Tutorial				x	
http://www.asic-world.com/verilog/veritut.html	Tutorial				x	
http://www.gmvhdl.com/VHDL.html	Tutorial			x		
http://www.doulos.com/knowhow/	Tutorial Information			x	x	x
http://www.cs.ucr.edu/content/esd/labs/tutorial/	Information			x		
http://www.xtrj.org/collection/vhdl.htm	Information			x		
http://instruct1.cit.cornell.edu/courses/ee475/tutorial/VHDLTut.htm	Tutorial Links			x		
http://www.systemc.org/	Information					x
http://www.acc-eda.com/vhdlref/refguide/vhdl_examples_gallery/vhdl_examples_gallery.htm	Examples			x		
http://www.vhdl.org/	Information			x	x	x
http://www.vhdl.org/vhdlsynth/vhdl_examples/	Examples			x		
http://www.acc-eda.com/vhdlref/refguide/toclist.htm	Information			x		

NASA Complex Electronics Guidebook for Assurance Professionals

http://www.mrc.uidaho.edu/fpga/index.php	Information	x				
http://www.fpga4fun.com/	Information	x				
http://equipe.nce.ufrj.br/gabriel/vhdlfpga.html	Links	x		x		
http://www.fuse-network.com/fuse/training/index.html	Training Material	x				x
http://www01.edatoolsafe.com/books/ASIC/ASICs.php	ASIC on-line book					x
http://aar400.tc.faa.gov/FlightSafety/TechReports/AR-95-31-CEH.pdf	FAA Tutorial	x				x
http://www.netrino.com/Articles/RCPimer/	Tutorial		x			
http://www.cotsjournalonline.com/home/	Journal					x
http://www.fpgajournal.com/	Journal	x				

Appendix B

B.1 CPLD



Description:

A Complex Programmable Logic Device (CPLD) contains a set of simpler Programmable Logic Device (PLD) blocks whose inputs and outputs are connected together by a global interconnection matrix. So a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed. A key feature of the CPLD architecture is the arrangement of logic cells on the periphery of a central shared routing resource. CPLDs are equivalent to about 50 typical PLD devices, and can replace thousands, or even hundreds of thousands, of logic gates

Programming and Reprogramming

CPLDs vary in how they can be programmed or reprogrammed, depending on their underlying structure. The three basic types of CPLDs are:

1. **Fuse or anti-fuse.** These CPLDs are programmed by passing a large current through the connections (fuses). The current “blows” the fuse to break a connection. The CPLDs are *one-time programmable* because you can't rewire them internally once the fuses are blown. Programming occurs in a special device external to the circuit board the CPLD will be placed on.
2. **EPROM or EEPROM.** In these CPLDs, the interconnections are made with transistors that are opened or closed by storing a charge on their gate electrodes using a high-voltage pulse. For EPROM-like CPLDs, you erase the CPLD and then place it in a special programmer socket and reprogram it. Reprogramming is not possible once the chip is soldered to its circuit board. EEPROM-like CPLDs may be reprogrammable on the circuit board, if special circuitry is included.

3. **SRAM or Flash.** Static RAM (SRAM) or Flash can be used to control the transistors for each interconnection. Each memory bit controls the interconnect switches through its value. When a bit is set to '1', the switch is closed, and the logic elements are connected. A '0' opens the switch. CPLDs built using RAM/Flash switches can be reprogrammed without removing them from the circuit board and are *in-circuit reconfigurable* or *in-circuit programmable*.

So how do you figure out what switches to open or close to implement your logic design? Fortunately, there are tools available to take that logic design and output a binary file which configures the switches in a CPLD.

Applications:

CPLDs are used in a wide variety of applications from cell phones to spacecraft. They are often used as “glue logic” to connect various parts of a design, massage and process data, or to translate data from one protocol to another.

CPLDs are great for:

- high speed operations
- interface controllers (bus, memory, Flash)
- interface bridging
- I/O expansion
- device configuration
- power-up sequencing
- microprocessor support logic
- glue logic
- implementing small “soft” microcontrollers (e.g., 8-bit)

They are used as support chips in most modern electronics, including:

- Cell phones
- PDAs
- Digital cameras
- Communications hardware
- GPS

CPLDs come in a variety of density, speed, and package options. Handheld applications tend to use lower density devices, because they have less need for complex logic, require low power, and try to minimize cost per unit. When capability is more important than power usage, higher density CPLDs are a better choice.

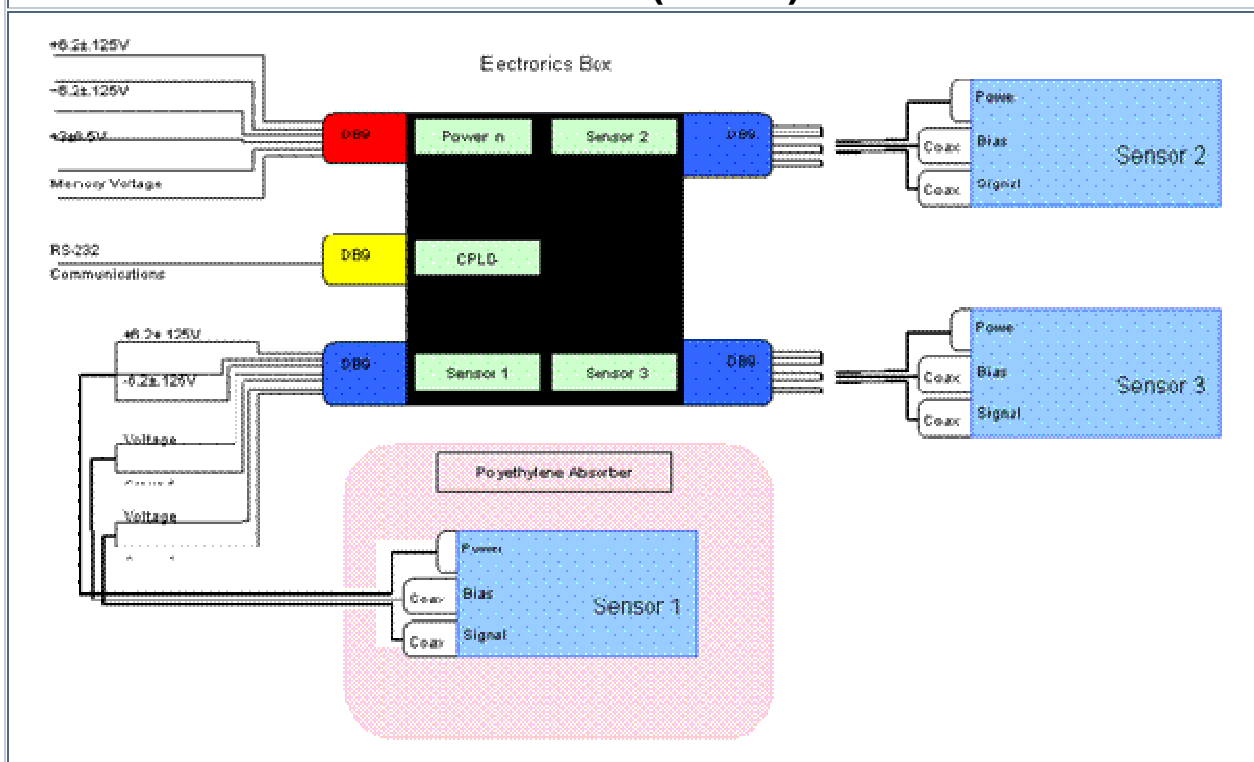
Often a logic design could be implemented in either a CPLD or an FPGA (Field Programmable Gate Array). CPLDs are chosen when predictable timing performance is required. CPLDs have fewer routing matrices than FPGAs. Since each routing matrix adds a little delay to the signal, fewer routings translates to faster signal transit. While CPLD density is less than most FPGAs,

high end CPLDs will have same density as low end FPGAs. Performance of CPLDs is usually better than FPGAs, though it depends on the vendor, size (number of cells), speed, and other factors.

Real-world Examples:

Here's some examples of CPLDs used in a variety of real products.

MicroDosimeter Instrument (MIDN)



MIDN is a space payload that will flight test a compact, low powered, and portable, solid-state micro dosimeter. MIDN will collect quantitative information on the dose and dose distribution of energy deposited in silicon cells that are tissue-sized. By inference, this data will show what the dosage would be in living tissue.

CPLDs are used in MIDN for command and data handling. This payload is part of the MidSTAR (Midshipman Space Technology Applications Research) satellite that is scheduled for launch in 2006.

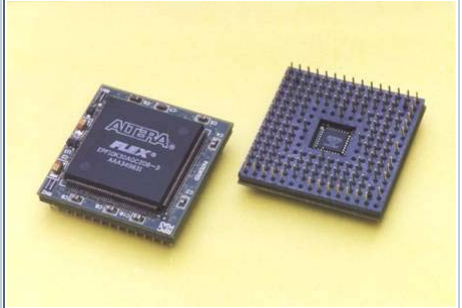
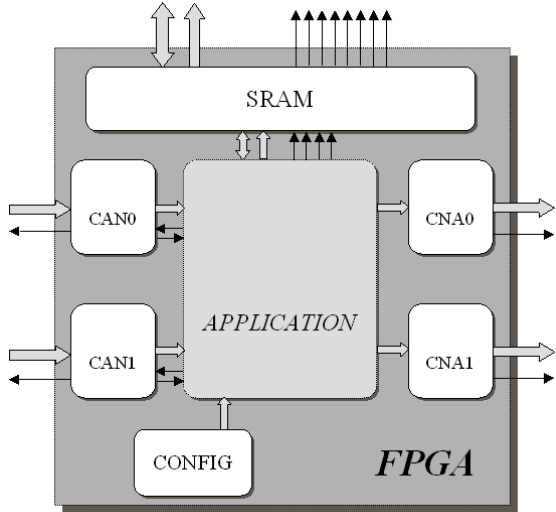
Fluids and Combustion Facility (FCF)



The Fluids and Combustion Facility (FCF) will be a permanent modular, multi-user facility to accommodate microgravity science experiments onboard the International Space Station (ISS) U.S. Laboratory Module. The Fluids Integration Rack (FIR) is used to perform fluid physics experiments in microgravity. The FIR will permit a wide range of fluid investigations from microscopic imaging to particle tracking. The Combustion Integration Rack (CIR) is used to perform combustion experiments in microgravity. Experiments will look at how solid, liquid, and gaseous fuels burn in microgravity, fire prevention and suppression, pollutant and particulate formation, and combustion efficiency.

CPLDs are used within FCF to translate data from a digital camera to a high-speed fiber interface. When the data is received, two other CPLDs reformat the incoming data to what is required by a Digital Signal Processor (DSP).

B.2 FPGA

Device name:	Field Programmable Gate Array (FPGA)
	

Description:

A Field Programmable Gate Array (FPGA) is a collection of simple, configurable logic blocks arranged in an array with interspersed switches that can rearrange the interconnections between the logic blocks. Each logic block is individually programmed to perform a logic function (such as AND, OR, XOR, etc.) and then the switches are programmed to connect the blocks so that the complete logic functions are implemented.

The interconnections for the logic blocks are programmable switches. FPGAs may use EEPROM, SRAM, antifuse, or Flash technology to store the programming. In most larger FPGAs the configuration is volatile, and must be re-loaded into the device whenever power is applied or different functionality is required.

Initially, FPGAs had only local and global routing resources (i.e., a logic block could only connect to adjacent logic blocks or to global networks). Newer FPGAs have multilevel routing hierarchies, so logic blocks can connect to different levels. Fortunately, the design software will take care of these complex issues for you.

Newer FPGAs are being developed that contain fixed functionality, as well as traditional programmable logic. FPGAs may contain a FIFO, arithmetic functions, memory, chip-to-chip transceivers, digital signal processor (DSP), or even an entire bus interface or microprocessor core. FPGAs with fixed functionality are cousins to the System-on-chip (SoC) devices that included programmable logic as part of their design.

SRAM FPGAs

SRAM, or static RAM, is a volatile type of memory. The contents of the memory are lost whenever the power is switched off. Static RAM differs from the dynamic RAM used in PCs in that memory refresh of the RAM is not required. SRAM-based programmable logic devices, such as FPGAs, have to be programmed every time the chip is switched on. This is usually done automatically by another part of the system.

Most SRAM-based FPGAs use a master mode, where they read the configuration information from non-volatile memory, such as a serial or parallel EPROM or flash memory. The FPGAs can also be configured via an external source in slave mode. The FPGA accepts a serial or parallel data stream that represents the configuration data. The source of the data can be a processor, computer, or an FPGA that is acting as a master. Using this technique, it's possible for several FPGAs to be programmed from a single memory. A master FPGA is wired to a daisy chain of slave FPGAs. When the master FPGA has been programmed, it will keep reading the data from the memory and pass it on to the slave devices until all of the FPGAs are configured.

Antifuse FPGAs

A fuse is a special part of the programmable chip that is normally closed (connected) until an electrical current breaks that connection. Antifuses, unlike traditional fuses, are open until a voltage is applied to close (complete) the circuit path. Once programmed closed, the connection cannot be reprogrammed to open. Programmable logic that uses fuses or antifuses are “program once” chips.

Antifuse FPGAs are best used when you don't want to have to reconfigure your chip every time power is applied (e.g. if you need a quick power-on time). They are also useful in environments where SRAM would have problems (e.g. high altitude or outer space).

Flash FPGAs

Flash memory is non-volatile, which means that it retains its contents even when the power is switched off. It can be erased and reprogrammed as required. This makes it useful for programmable logic device memory. Flash-based devices combine the best of both worlds - maintaining configuration when not powered, but also allowing reprogramming when desired. Flash-based programmable devices are essentially immune to neutron radiation (generated when cosmic rays interact with the atmosphere) and are resistant to other high-energy particles.

Software Engineers and FPGAs

What if a software engineer could create a regular software application that could run on an FPGA? Design tools for FPGAs are moving quickly in this direction. In this new environment for software developers, the FPGA can be viewed as one possible target (along with traditional and non-traditional processor architectures) for a software compiler. With currently available

tools, the software engineer can make use of FPGA platforms, as well as take advantage of the high level of algorithmic parallelism that is available when traditional processors (or processor cores) and FPGAs are combined in a single target platform.

FPGA-based computing platforms, particularly those with embedded “soft” microprocessors, have the potential to implement extreme high-performance applications. With the latest generation of hardware/software codesign tools it is now possible to use multiple graphical, software-oriented design methods as part of the FPGA design process.

Radiation and FPGAs

NASA projects typically deal with environments more extreme than an office or laboratory. Spacecraft and high-altitude aircraft are bombarded with radiation. Shock and vibration, electromagnetic interference, and thermal issues are problems commonly faced when designing NASA systems.

Unfortunately, FPGAs are mostly just big RAM devices, and most of that RAM is in the configuration circuitry. An upset event in the routing can quietly alter the logical interconnections and a problem in a lookup table (LUT) can alter the functional behavior of a design.

SRAM FPGAs are susceptible to ionizing radiation, including the neutron radiation experienced at high altitudes. SRAM FPGA designed for high-radiation environments typically use periodic read-back and verification of the configuration or frequent reconfiguration of the chip to a known good state. Because SRAM devices are vulnerable, they are used more in “payload” applications, where some level of failure can be tolerated and overcome, instead of in the more critical systems that control spacecraft flight operations.

While antifuse FPGAs lag behind the more programmable versions in size (gate density), versatility and market share, they are very useful in space applications. Radiation tolerant FPGAs use the antifuse technology, which provides immunity to radiation effects as well as low power, single-chip solutions that do not require configuration circuitry.

Flash-based FPGAs provide radiation tolerance along with reprogrammability. Like antifuses FPGAs, they are immune to upsets caused by most radiation. Like SRAM FPGAs, they can be reprogrammed in-circuit. Radiation studies of flash-based FPGAs are still on-going.

While high-profile projects like the Mars rovers showcase the use of programmable logic in space, the majority of space-bound FPGAs are included in commercial and military satellites. FPGAs are frequently used in satellite functions such as guidance, station-keeping, and telemetry.

Applications:

FPGAs had an initial niche as prototypes for Application-Specific Integrated Circuits (ASIC). Because ASICs require a long lead time from design to implementation, and it can be very

expensive to correct ASIC design errors, FPGAs were used to try out the designs. Errors detected in the design could be corrected, the FPGA reprogrammed, and testing of the design could continue. The process isn't without problems, though. ASIC designs had to be created using ASIC synthesis tools, then a separate FPGA tool is used to implement the ASIC prototype in an FPGA. Switching from one synthesis tool to another requires changing code and scripts, which is time-consuming, and increases the potential for introducing errors into the prototype that do not accurately reflect the functionality of the ASIC design. FPGAs are often slower than ASICs, which prevents timing problems from being accurately diagnosed. Despite the problems, however, FPGAs are still used to prototype ASICs - because the cost of a failed ASIC can be quite expensive.

FPGAs have gained rapid acceptance and growth over the past decade because they can be applied to a very wide range of applications. Some typical applications are:

- random logic
- integrating multiple SPLDs
- device controllers
- bus controllers
- communication encoding and filtering
- small to medium sized systems with SRAM blocks

More intensive applications include:


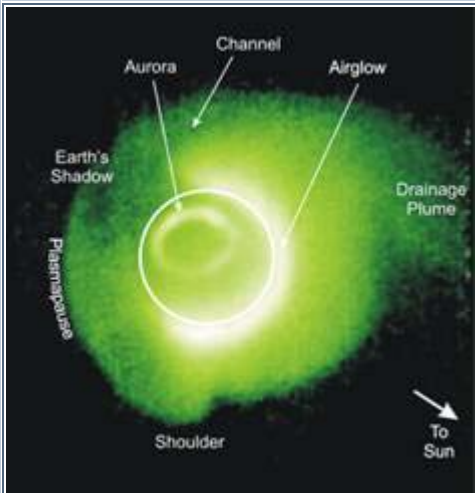

- Digital signal processing
- Complex custom applications
- Consumer electronics
- Software radio
- Cryptographic and security devices

Reconfigurable or adaptive computing is a cutting-edge application for FPGAs. Instead of a traditional microprocessor that executes software, FPGAs are reprogrammed to perform the necessary calculations or operations.

NASA (and other) Examples:

The Mars Exploration Rovers (MERs), Spirit and Opportunity, garnered the world's attention as they rolled out onto the surface of Mars. Hidden inside the rovers and landers are FPGAs, doing their job in a harsh environment. FPGAs are used in pyrotechnics devices for landing, as well as in the arm, cameras, steering, antenna gimbals, and wheel control systems on the Mars rover missions.

Here's some other space and science related projects that use FPGAs:

<i>Cassini</i>	
	There are also FPGAs orbiting Saturn on the Cassini spacecraft. FPGAs are used in many instruments on Cassini, including the Visual and Infrared Mapping Spectrometer (VIMS).
<i>Extreme Ultraviolet Imager (EUV)</i>	
	FPGAs control parts of the EUV instrument on the IMAGE (Imager for Magnetopause-to-Aurora Global Exploration) satellite. FPGAs control the sensors and read out, format, and store the data.
<i>Optus C1</i>	
	Radiation tolerant FPGAs have been deployed on board Optus C1, the largest hybrid commercial and defense communications satellite ever launched. The communications satellite was launched in 2003 for Australian Defense Force.

- A prototype multi-directional muon detector, operating in Sao Martinho, Brazil, is being upgraded and extended, using FPGAs. The FPGAs allow a more complicated and advanced logical circuit to be designed at a reduced cost. The upgraded detector will be able to determine the incident direction of every single muon detected and record the count rates in the total 121 incident directions.

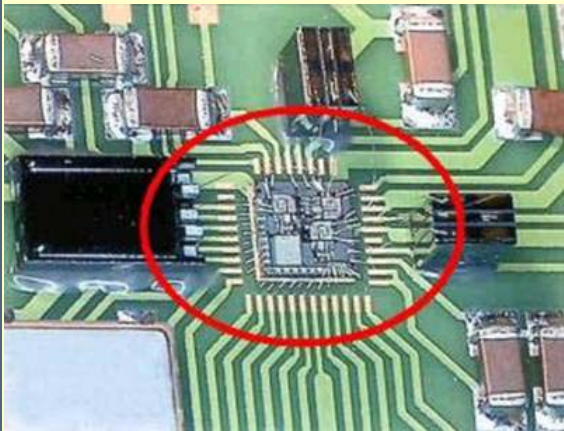
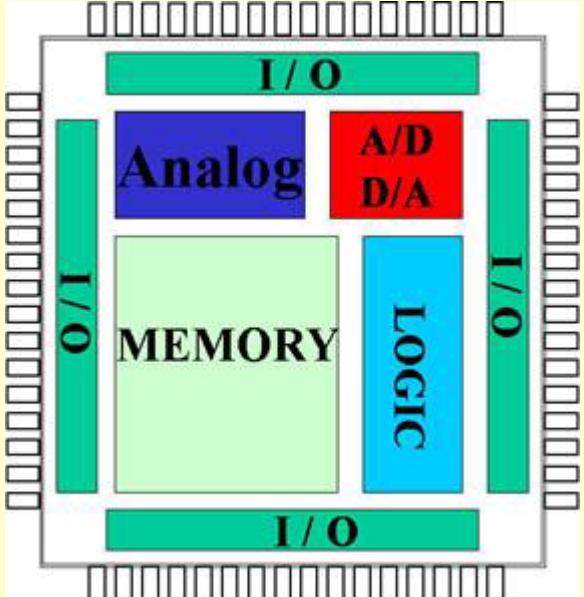
NASA Complex Electronics Guidebook for Assurance Professionals

- NASA's Jet Propulsion Laboratory has developed a lossless image-compression algorithm that can be implemented entirely in an FPGA plus a small random-access memory chip.

Other missions that include FPGAs:

- Civilian/Scientific exploration:
 - Deep Space 1
 - Mars Pathfinder, Surveyor, Express, Climate Orbiter
 - Lunar Prospector
 - SIRTf
 - TDRS
 - Hubble Space Telescope
 - GOES
- International Missions
 - International Space Station
 - Chandra
 - Rosetta
 - SOHO
- Commercial Satellites
 - Telstar
 - PanAm Sat
 - Intelstat IX
 - Globalstar
 - Orbview
- Military Satellites
 - Clementine
 - HESSI
 - Mighty Sat
 - SBIRS-High (-Low)
- Launch Vehicles
 - - Ariane
 - Atlas
 - Delta
 - EELV
 - SeaLaunch

B.3 ASIC

Device name:	Application Specific Integrated Circuit (ASIC)
	

Description:

An Application-Specific Integrated Circuit (ASIC) is an integrated circuit designed to perform a particular function by defining the interconnection of a set of basic circuit building blocks drawn from a library provided by the circuit manufacturer. They are built by connecting existing circuit building blocks in new ways. Since the building blocks already exist in a library, it is much easier to produce a new ASIC than to design a new chip from scratch.

ASICs are custom-designed integrated circuits, but they are not programmable by the user. They are manufactured (usually in large quantities) by vendors according to the design provided by the customer. If you find a problem with an ASIC after it's produced, the only option is to remanufacture (re-spin) the chip with a corrected device. To avoid costly mistakes, FPGAs are often used to check out and debug the ASIC design prior to submittal to the manufacturer.

While most integrated circuits (ICs) could be considered “application-specific”, because they have a defined purpose, off-the-shelf parts are not really ASICs. They are not designed by the user/customer to incorporate just the required functionality. Examples of ICs that are not ASICs include standard parts such as memory chips (ROMs, DRAM, and SRAM), microprocessors, and all the miscellaneous chips that are used in modern electronics (FIFOs, logic chips, drivers, clock chips, switches, etc.). Now, if a chip has been designed specifically for a talking toy, a cell phone, or a satellite, it's an ASIC. As a general rule, if you can find it in a data book, then it is probably not an ASIC.

Integrated Circuits are made on a thin (a few hundred microns thick), circular silicon wafer, with each wafer holding hundreds of die. The transistors and wiring are made from many layers built on top of one another. Each successive layer has a pattern that is defined using a mask similar to a glass photographic slide. The first layers define the transistors, and the last layers define the metal wires between the transistors (the interconnections).

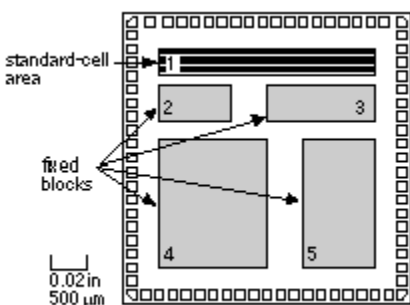
ASICs come in two basic varieties - full-custom and semi-custom, which consists of two sub-types: cell-based and gate-array. Each type of ASIC has strengths and weaknesses. A microprocessor is an example of a full-custom ASIC, where each micron on the silicon is customized to give exactly what is needed. Semi-custom ASICs have pre-designed elements and customizable portions.

A full-custom ASIC allows customization of some (and possibly all) logic cells and all mask layers. Customizing all of the ASIC features in this way allows designers to include analog circuits, optimized memory cells, or mechanical structures on an IC, for example. Full-custom ASICs are the most expensive to design and manufacture. The manufacturing lead time (how long it takes to make an ASIC once the design is completed) is typically eight weeks.

Semi-custom ASICs have all of the logic cells pre-designed and some (possibly all) of the mask layers are customized. Designers use pre-designed cells from a cell library, provided by the vendor or a third party. These pre-designed units are usually referred to as IP (Intellectual Property). Semi-custom ASICs are either standard cell-based ASICs or gate-array-based ASICs.

A cell-based ASIC uses pre-designed logic cells (e.g., AND gates, OR gates, multiplexers, and flip-flops) known as standard cells. The standard-cell areas (also called flexible blocks) are built of rows of standard cells-like a wall built of bricks. The standard-cell areas may be used in combination with larger pre-designed cells, such as microcontrollers, known as megacells. Megacells are also called megafunctions, full-custom blocks, system-level macros (SLMs), fixed blocks, cores, or Functional Standard Blocks (FSBs).

The ASIC designer defines only the placement of the standard cells and interconnect in a cell-based ASIC. However, the standard cells can be placed anywhere on the silicon; this means that all the mask layers are customized and are unique to a particular customer. The advantage of cell-based ASIC is that designers save time, money, and reduce risk by using a pre-designed, pre-tested, and pre-characterized standard-cell library. In addition each standard cell can be optimized individually.



If you were to look through a low-powered microscope at a cell-based ASIC die, you would see something similar to this figure. This ASIC has a single standard-cell area (a flexible block) together with four fixed blocks. The small squares around the edge of the die are bonding pads that are connected to the pins of the ASIC package.

In gate-array-based ASICs, the transistors are predefined on the silicon wafer. This predefined pattern of transistors on a gate array is called the base array, and the smallest element that is replicated to make the base array is called the base cell. Only the top few layers of metal, which define the interconnect between transistors, are defined by the designer using custom masks.

The designer chooses from a gate-array library of pre-designed and pre-characterized logic cells or macros. The reason for this is that the base-cell layout is the same for each logic cell, and only the interconnect (inside cells and between cells) is customized. Gate-array ASICs can be prefabricated up to a point and stored. At a later time, the final customization steps can be performed to complete the ASIC. This reduces the manufacturing time to only a few days or at most a couple of weeks.

ASIC Cell Libraries

The cell library is the key part of ASIC design. Cell libraries can be provided by the ASIC vendor, procured from a third-party library vendor, or custom-built. The first choice, using an ASIC-vendor library, requires you to use a set of design tools approved by the ASIC vendor to enter and simulate your design. An ASIC vendor library is normally a phantom library - the cells are empty boxes, or phantoms, but contain enough information for layout. After you complete layout you hand off a netlist to the ASIC vendor, who fills in the empty boxes (phantom instantiation) before manufacturing your chip.

The second and third choices require you to make a buy-or-build decision. If you complete an ASIC design using a cell library that you bought, you also own the masks (the tooling) that are used to manufacture your ASIC. This is called customer-owned tooling (COT, pronounced “see-oh-tee”). A library vendor normally develops a cell library using information about a process supplied by an ASIC foundry. An ASIC foundry (in contrast to an ASIC vendor) only provides manufacturing, with no design help. If the cell library meets the foundry specifications, we call this a qualified cell library. These cell libraries are normally expensive (possibly several hundred thousand dollars), but if a library is qualified at several foundries this allows you to shop around for the most attractive terms. This means that buying an expensive library can be cheaper in the long run than the other solutions for high-volume production.

The third choice is to develop a cell library in-house. Many large computer and electronics companies make this choice. Most of the cell libraries designed today is still developed in-house despite the fact that the process of library development is complex and very expensive.

However created, each cell in an ASIC cell library must contain the following:

- A physical layout
- A behavioral model
- A Verilog/VHDL model
- A detailed timing model
- A test strategy
- A circuit schematic
- A cell icon

- A wire-load model
- A routing model

Applications:

ASICs are used widely in many types of electronics devices. Anytime there are a large number of devices manufactured that require specialized operation, you will probably find an ASIC inside.

ASICs Application Examples:

- Battery management for household appliances
- Low noise audio circuit
- Analog ASIC for industrial environment
- Sensitive photo transistors and opto-sensors
- DC-DC converters from 0.9V supply voltage
- Control circuit for cycle rear light
- 120V Linear Regulator
- Interface circuit for a bar code reader
- Control and evaluation circuit for motion detectors
- Timer electronics
- Interface and signal processing electronics for sensors (light, vibration and magnetic field)
- Control circuit for mobile phones
- Automotive control functions
- PDAs.

NASA Examples:

ASICs can provide several features that are especially important in spacecraft and instruments, such as:

- Customized electronics
- Smaller footprint
- Less weight
- Hard-coded (radiation resistant)

The smaller footprint on the circuit board and reduced weight are the result of including multiple functions in a single chip, rather than having to use many individual integrated circuit chips.

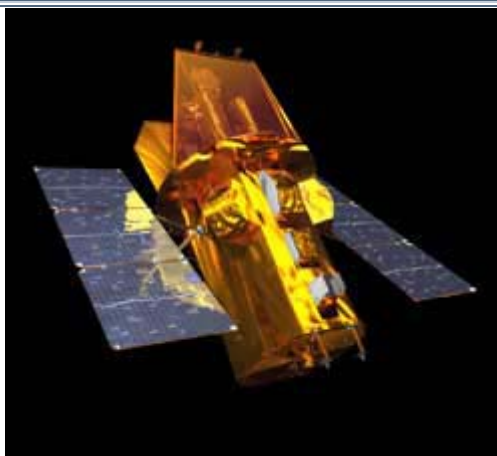
Cassini



The Cassini spacecraft is in a four-year tour to study Saturn, its rings, moons and magnetosphere. The spacecraft is a complicated system, with 22,000 wire connections and nearly nine miles of cabling. The main on-board computer uses very high-speed ICs and advanced, radiation-hardened ASICs. Each ASIC replaces one hundred or more traditional chips, allowing the development of a data system for Cassini that is ten times more efficient than earlier spacecraft designs (e.g., Galileo and Magellan), but at less than one-third the mass and volume. Mars Pathfinder and Near Earth Asteroid Rendezvous (NEAR) both used these chips directly off the Cassini production line.

The Cassini program also created an advanced solid-state power switch that eliminates the rapid fluctuations (called transients) usually found in circuits utilizing conventional power switches. This power switch combined the switching attributes of the Metal-Oxide Semiconductor Field-Effect Transistor (MOS FET) with an ASIC design. This ASIC results in significantly improved component lifetime and efficiency, and is widely applicable to both industrial and consumer electric and electronic products.

Swift

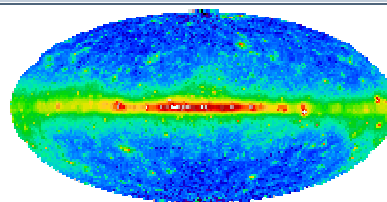


Swift is a first-of-its-kind multi-wavelength observatory dedicated to the study of gamma-ray burst (GRB) science. Its three instruments will work together to observe GRBs and afterglows in the gamma ray, X-ray, ultraviolet, and optical wavebands. The main mission objectives for Swift are to:

- Determine the origin of gamma-ray bursts
- Classify gamma-ray bursts and search for new types
- Determine how the blastwave evolves and interacts with the surroundings
- Use gamma-ray bursts to study the early universe
- Perform the first sensitive hard X-ray survey of the sky

One instrument on Swift is the Burst Alert Telescope (BAT), a large coded aperture instrument with a wide field-of-view (FOV) that provides the gamma-ray burst triggers. BAT will observe and locate hundreds of bursts per year to better than 4 arc minutes accuracy. BAT contains thousands of detector elements that are assembled into 8 x 16 arrays, each connected to 128-channel readout ASICs.

Gamma-ray Large Area Space Telescope (GLAST)

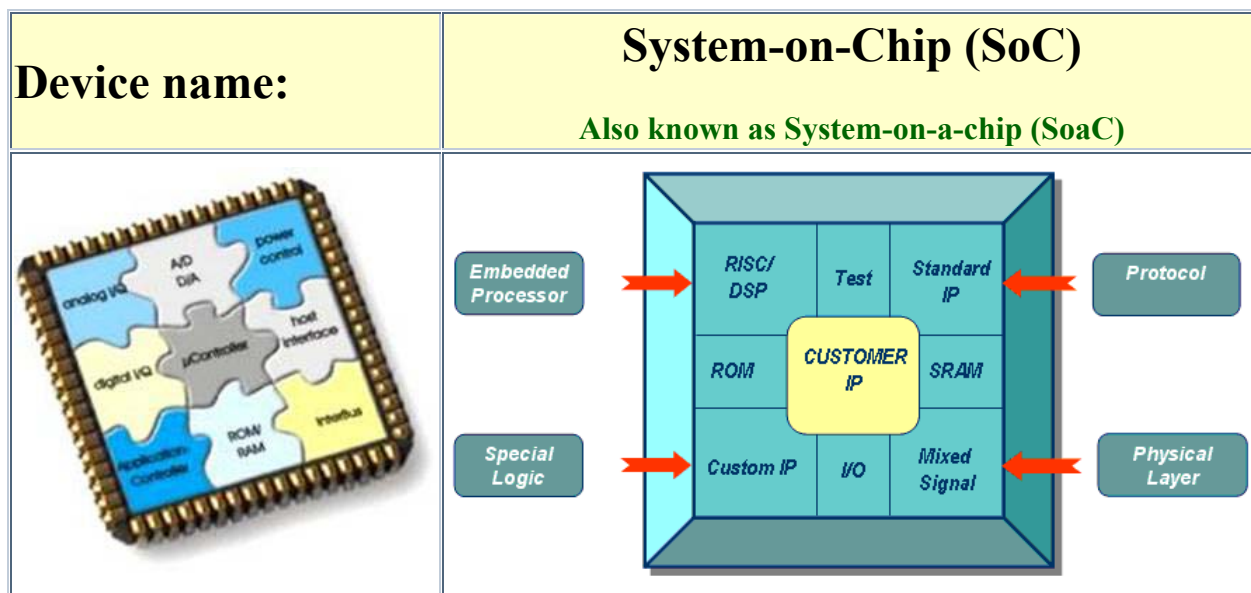


The Gamma-ray Large Area Space Telescope (GLAST) is an international and multi-agency space mission that will study the cosmos in the energy range 10 keV - 300 GeV. GLAST will have an imaging gamma-ray telescope vastly more capable than instruments flown previously, as well as a secondary instrument to augment the study of gamma-ray bursts. The main instrument, the Large Area Telescope (LAT), will have superior area, angular resolution, and field of view over previous instruments. The LAT tracker subsystem was concerned about compactness, minimum wiring, and redundancy. The subsystem was implemented using two ASICs.

SonoSite's TITAN™ system took part in a 10-day underwater experiment with NASA Extreme Environment Mission Operations (NEEMO) 7 Mission. Aquanauts used the laptop-sized ultrasound system to scan each other in simulated emergency situations and transmit live images to a hospital for review by radiologists. The TITAN system utilizes SonoSite's proprietary ASIC microchip technology to integrate millions of transistors onto one circuit.

NASA's Jet Propulsion Laboratory has developed a command interface ASIC and an analog interface ASIC. This chip set for remote actuation and monitoring of a collection of switches can be used to control generic loads, pyrotechnic devices, and valves in a high-radiation environment. The command interface ASIC (CIA) can be used alone or in combination with the analog interface ASIC (AIA). Designed primarily for incorporation into spacecraft control systems, they are also suitable for use in high-radiation terrestrial environments (e.g., in nuclear power plants and facilities that process radioactive materials).

B.4 SoC



Description:

System-on-chip (SoC; also called “system-on-a-chip” or SoaC) is a complete product that contains all the necessary electronic circuits and parts for a "system" on a single integrated circuit. Think of it as a single-board-computer on a chip. SoCs include the hardware components and all required ancillary electronics.

SoCs combine aspects of ASICs and field-programmable logic. SoCs can be:

1. Totally ASIC, with the individual blocks specified by the designer
2. ASIC for the computing unit and logic functions, with some programmable parts (e.g., CPLD)
3. Implemented on programmable logic (e.g., FPGA)

SoCs can use “IP” (Intellectual Property) – designs created by others and integrated into the chip. IP blocks are pre-designed behavioral or physical descriptions of a standard component. These reusable components are usually Commercial-off-the-Shelf (COTS) products.

The benefits of SoC design include:

- Conservation of space (reduction in chip count)
- Improved performance (higher reliability)
- Lower memory requirements
- Greater design freedom (simpler logistics)

These benefits also come with some challenges including:

- Larger design space
- More expense (global on-chip communication is expensive in terms of power/propagation delay)
- Increased prototype cost
- Correctness of complete system with multiple components
- A high level of debugging methodology

Testing of the products is also a challenge due to the fact that typical testing methods have been developed for specific specialty areas, whereas the SoC requirement includes all specialties, potentially on one platform.

A SoC could include:

- Microprocessor
- Memory (e.g., SRAM, DRAM, Flash)
- Communications cores
- Digital Input/Output functions
- Analog Input/Output functions
- Bus controllers (e.g., PCI)
- DSP (Digital Signal Processor)
- Sensors
- Programmable logic (e.g., FPGA, CPLD)
- Embedded software

For example, a system-on-chip for a sound-detecting device might include an audio receiver, an analog-to-digital converter (ADC), a microprocessor, necessary memory, and the input/output logic control for a user - all on a single microchip

Configurable System-on-Chip (CSoC)

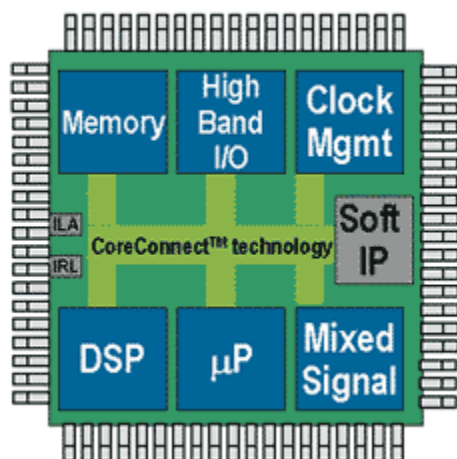
Configurable SoCs are a new form of system-on-chip that has a configurable fabric that designers can manipulate, after chip fabrication, to achieve specific functionality. Configurability lets you change on-chip functions for a variety of reasons. These reasons include:

- change in core functionality
- compatibility with a change in a communications or other standard to which the CSoC must conform
- correcting a design error incurred during original chip development.

Post-process configurability lets you create products that can adapt to changing requirements.

Some configurable SoCs are FPGAs that combine both hard (fixed) and soft (programmable) cores. These chips are sometimes referred to as platform FPGAs. In the diagram below, the microprocessor is a hard component (fixed in the silicon), while the Digital Signal Processor (DSP) is a soft component created in the FPGA programmable infrastructure.

The reconfigurable approach offers significant advantages. It reduces design costs because changes can be made immediately to the chip during development. Chip simulation becomes less of an issue because the real hardware is available immediately. In the field, bug fixes and upgrades can be more extensive as significant portions of the hardware can be altered, not just the application code.



Cost is the main downside to using a standard reconfigurable SoC rather than creating a custom SoC. Custom designs typically have large up-front development costs, but low individual chip costs. Reconfigurable SoCs have a comparatively small up-front cost, but are usually more expensive per chip. Reconfigurable SoCs can also be used for prototyping because the core CPU and fixed peripherals are well defined. Building a custom ASIC or SoC based on a reconfigurable prototype is relatively easy.

Applications for SoC:

System-on-chip devices can be used in any application that requires a processor and peripheral components. Since the advantages of SoC is small size, integrated components, and reduced power, they are especially useful in:

- Cell/camera phones
- Medical equipment (especially portable devices)
- Portable multimedia devices
- Network-enabled devices
- PDAs
- Point-of-sale devices
- Gaming systems

In the medical world, portable equipment and implantable devices are becoming more common. Such equipment includes blood glucose monitoring systems, insulin pumps, body temperature sensors, defibrillators, neurological stimulators, pacemakers and hearing aids. These products not only simplify the testing, monitoring, and treatment processes, but can also help to improve the quality of life for the patient by minimizing time spent in hospitals and often providing automatic, continuous treatment of chronic conditions.

NASA Complex Electronics Guidebook for Assurance Professionals

To address requirements for performance, power consumption, and size, medical equipment manufacturers are incorporating as much functionality as possible into a single, complex system-on-chip (SoC). These devices need to integrate both analog and digital capabilities and, in many cases, deliver short-range, low-data-rate wireless communications functionality. Furthermore, some applications may also require that high-voltage output stages be integrated into the same device. A variety of semiconductor technologies, intellectual property (IP) blocks, and support tools can help to significantly simplify the implementation of SoCs for implantable and portable medical devices.

An example of a network device is the Sony Video Network Station. This device, which contains an embedded Linux operating system running on an Axis ETRAX system-on-chip processor, transmits images generated by analog video cameras to remote locations where they can be viewed using ordinary GUI-based web browsers. The device is useful in a diverse range of applications requiring remote video monitoring and control, including security monitoring, quality inspection, image distribution, access control, and market research.

NASA projects:

Like ASICs and FPGAs, System-on-Chip (SoC) devices have significant benefits for NASA projects, including:

- Customizable electronics
- Smaller circuit board footprint
- Less weight
- Integrated functionality

Temperature Remote I/O (TRIO) System-on-Chip for Aerospace

The TRIO smart sensor data acquisition chip was developed by John Hopkins University/Applied Physics Laboratory for NASA spacecraft applications. TRIO includes a 10 bit self-corrected analog-to-digital converter, analog inputs, a front end multiplexer with selectable acquisition time, a current source, memory, serial and parallel bus, and control logic. These functions are very useful for spacecraft and subsystems health and status monitoring, and control actions. The key contributions of the TRIO are feasibility of modular architectures, elimination of several miles of wire harnessing, and power savings by orders of magnitude. So far TRIO is used in many missions including Contour, Messenger, Stereo, Europa Orbiter, Mars Surveyor Program, Solar Probe, Pluto Express, and the generic JPL X2000 spacecraft bus.

Radio Frequency (RF) components

Micro-Electro-Mechanical Systems (MEMS) is the integration of mechanical elements, sensors, actuators, and electronics on a common silicon substrate through microfabrication technology. Microelectronic integrated circuits can be thought of as the "brains" of a system and MEMS augments this decision-making capability with "eyes" and "arms", to allow microsystems to sense and control the environment.

NASA Complex Electronics Guidebook for Assurance Professionals

NASA Glenn Research Center is developing microwave MEMS devices that integrate with miniature microwave (RF) transmission lines and components to build low loss RF distribution networks for System on a Chip (SOAC) and phase array antennas. These novel, low loss, miniature RF components will be fabricated using multilayer processing, and they will be combined with SOAC technology being developed by the University of Michigan and the JPL Center for Integrated Space Microsystems (CISM) for nano-sized science craft.

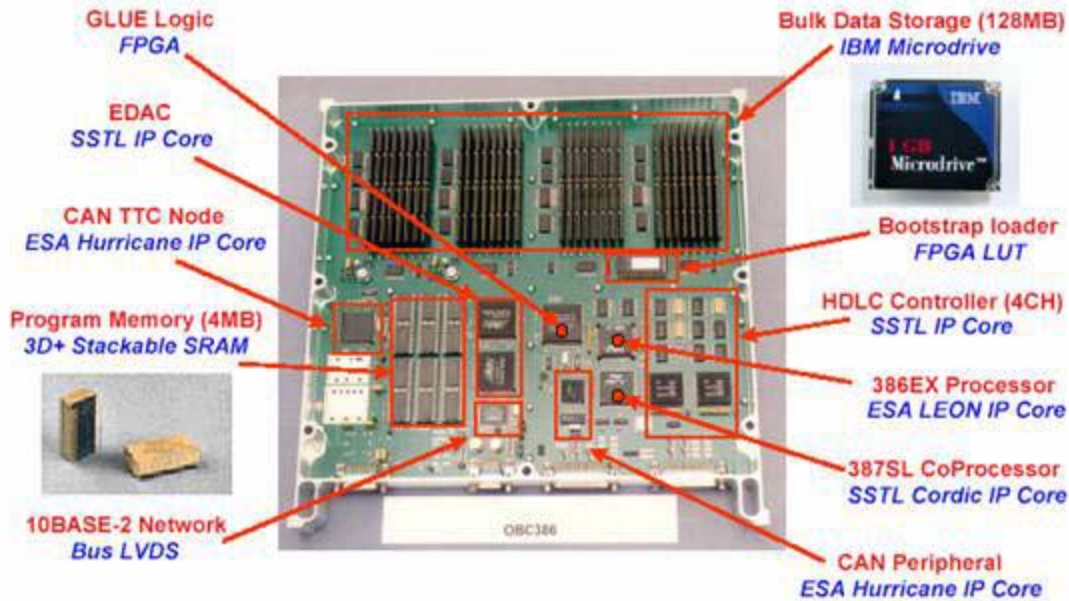
Advanced time-of-flight system-on-chip for remote sensing instruments

Accurate and/or fast time interval measurement is important in many remote sensing instruments, especially those that require detection of photon/particle events, position decoding and time-of-hit measurement. An advance time-of-flight (TOF) system-on-chip has been developed that includes the complete signal processing electronics for microchannel plate (MCP) readout. The TOF chip is capable of a time resolution of <50picoseconds. The TOF chip flies on the NASA/IMAGE spacecraft launched in 2000 and are part of many other science instruments on MESSENGER.

ChipSat

ChipSat is a long-term research program which aims to build a satellite-on-a-chip. As part of the program, an existing on-board computer (OBC) was scaled down to a system-on-a-chip (SoC). The OBC chosen was developed by the Surrey Satellite Technology Limited (SSTL), a company owned by the University of Surrey in Guildford, UK. The SoC is prototyped on a single high-density programmable logic array chip using soft intellectual property (IP) cores.

The image below shows the parts of the OBC that were mapped into the system-on-chip. An entire board was shrunk down to a single chip. The experiment showed that it is possible to implement the functionality of a small satellite OBC on a single programmable logic chip.



SCOC – A Spacecraft Controller On a Chip

The European Space Agency (ESA) is pursuing development of a system-on-chip that incorporates all the required functions for spacecraft control. This SoC is currently prototyped in an FPGA. The demonstration board is named BLADE (Development of the Board for LEON and Avionics DEMonstration). Eventually, the design will be produced in a radiation-tolerant ASIC or PROM-based FPGA.

SCOC looks to integrate multiple functions into a single chip. By integrating the functions, the external connections become on-chip interconnects. Other benefits include reduced power consumption, reduced component count (and thus lower mass), and increased performance and reliability. However, putting all the functions on a single chip reduces the accessibility to the internal functions, and makes testing the complex chip more difficult.

ESA would like the SCOC to include the following components

- Standard processor, known to the space community (the LEON SPARC-V8)
- Flexible peripherals, which can be powered down
- Telecommand and Telemetry (TM/TC) functionality (using the CCSDS protocol)
- Housekeeping and CCSDS Time Management
- *Multiple standard interfaces*
 - PCI parallel backbone
 - Spacewire (IEEE 1355.1)
 - MIL 1553 standard Bus Controller/Monitor (BC/BM) and Remote Terminal (RT)

- *Dedicated data processing*
 - Monitoring camera interface and image compression
 - GNSS navigation receiver
 - Star tracker pre-processor
 - Mathematical co-processor

The current BLADE development integrates the processor with standard interfaces. Additional functionality will be added in the future.

JPL Center for Integrated Space Microsystems (CISM)

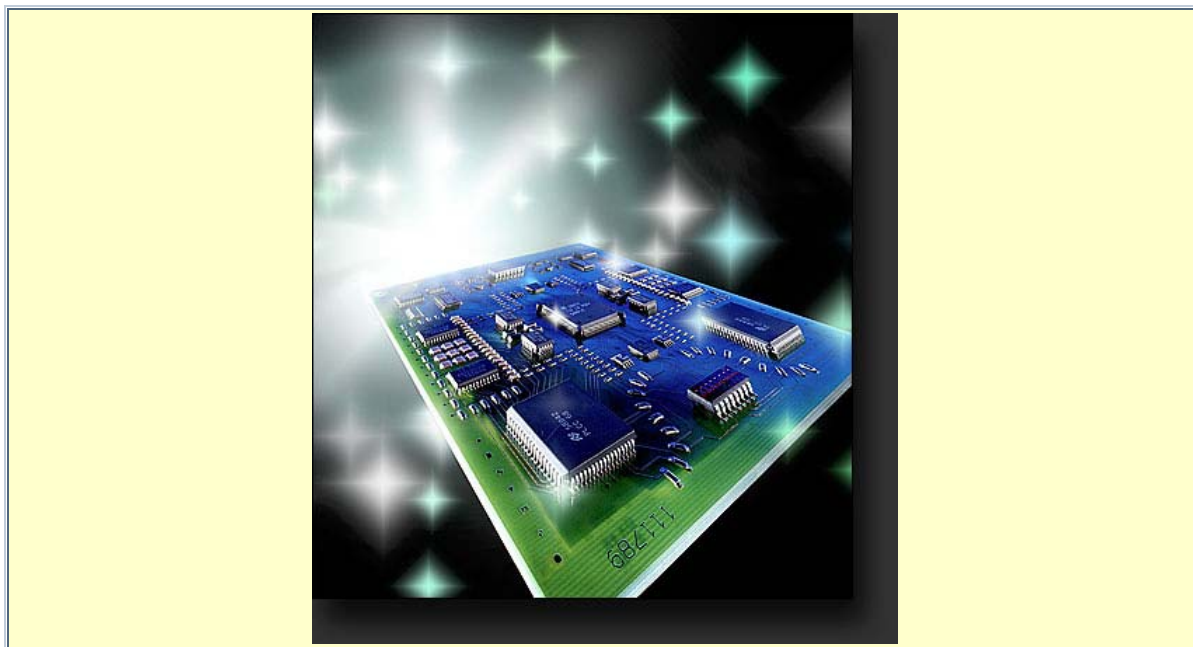
The Center for Integrated Space Microsystems is a NASA Center of Excellence for the advanced research and development of highly integrated microelectronics systems for deep space exploration. The scope of technologies includes:

- Optical, radio frequency (RF), and digital telecommunications processing
- Power management and distribution
- Data processing and storage
- Spacecraft command, control, and navigation
- Integrated sensors technology

The end result of this research and development is an advanced set of System On A Chip devices. CISM is working on highly integrated and autonomous, intelligent microavionics systems. The primary customer of CISM is the Deep Space Systems Program. Missions that will use this technology are planned for launch in 2008-2020, and include the Europa Lander, Titan Explorer, and Comet Nucleus Sample Return.

B.5 Reconfigurable Computing

Device name:	Reconfigurable Computing AKA Adaptable Computing, Evolvable computing
---------------------	---



Description:

Can you have a computer without a microprocessor? How do you deal with situations where autonomous instruments have to adapt to changing situations? What if your device has to support multiple protocols, depending on its location or mission? How do you process signals that may come in multiple formats “on the fly”?

The answer to the above questions is “reconfigurable computing”. Reconfigurable Computing represents a new idea in computing philosophy, in which some general-purpose hardware agent is *configured* to carry out a specific task, but can be reconfigured on-demand to carry out other specific tasks.

Traditionally, there have been two ways to implement a computation or algorithm: custom hardware or software. In some systems, this decision can be made on an individual subtask basis, placing some subtasks in custom hardware and some in software on more general-purpose processing engines.

Hardware designs offer high performance because they are:

- Customized to the problem—no extra overhead for interpretation or extra circuitry capable of solving a more general problem.
- Relatively fast, due to their highly parallel and spatial execution.

Software implementations exploit a “general-purpose” execution engine (i.e., microprocessor) which interprets a designated data stream as instructions telling the engine what operations to perform. As a result, software is:

- flexible—task can be changed simply by changing the instruction stream

- relatively slow—due to mostly temporal, serial execution
- relatively inefficient—since operators can be poorly matched to computational task.

Reconfigurable computing combines the best of both implementations, allowing general purpose software to be implemented in hardware. This class of architectures permits the computational capacity of the system to be highly customized to the instantaneous needs of an application, while also allowing the computational capacity to be reused in time at a variety of time scales.

The usual hardware agent for reconfigurable computing is a set of Field Programmable Gate Arrays (FPGAs). Reconfigurable computing manipulates the logic within the FPGA at run-time. The design of the hardware may change in response to the demands placed upon the system while it is running. Here, the FPGA acts as an execution engine for a variety of different hardware functions — some executing in parallel, others in serial — much as a microprocessor executes a variety of software threads.

Reconfigurable computing offers several advantages over custom hardware and general-purpose software implementations, including:

- Flexibility - the system can be changed as necessary, on the fly
- Simpler hardware design - you don't need a fancy high-powered microprocessor, just one or more FPGAs
- Speed - implementing algorithms in hardware results in faster execution, due to the parallel nature of hardware

The reconfigurable computing systems built during the last years have often achieved the performance several orders of magnitude higher than the traditional processor based solutions. Reconfigurable computing is now breaking into the commercial market in the areas of application-specific systems and information appliances, which include emerging areas like mobile communication, multimedia-based networks, encryption, and image processing.

What hardware is reconfigurable?

Not all FPGAs can be used in reconfigurable computing. User-configurable FPGAs can be programmed and reprogrammed by the user in a lab, or even in the field. But they cannot be dynamically reprogrammed as the system is running. Many older FPGAs read their configuration out of a serial EEPROM, and only when a chip reset signal is asserted. This means that the FPGA must be reprogrammed in its entirety and that its previous internal state cannot be captured beforehand.

In order to benefit from run-time reconfiguration, the FPGAs involved need some or all of the following features, which increase design flexibility:

- **On-the-fly reprogrammability.** Resetting the FPGA takes a lot of time and should be avoided whenever possible.

- **Partial reprogrammability.** The ability to leave most of the internal logic in place and change just one part is an important factor in reconfigurable systems. It will always be much faster to change a small piece of the logic than the entire FPGA contents.
- **Externally-visible internal state.** If you can see the internal state of the FPGA at any time, then it is also possible to capture that state and save it for later use. This allows the internal state of the FPGA to be read and written just like memory or processor registers and makes it possible to “swap” hardware designs in much the same way that pages of virtual memory are swapped into and out of physical memory.

Run-time environments

How does the reconfigurable system know what to do at any given time? That job is usually handled by software. The software manages the processes of:

- deciding which hardware objects to execute and when
- swapping hardware objects into and out of the reconfigurable logic
- performing routing between hardware objects or between hardware objects and the hardware object framework.

Of course, having software manage the reconfigurable hardware usually means having an embedded processor or microcontroller on-board. The embedded software that runs there is called the run-time environment and is analogous to the operating system that manages the execution of multiple software threads. Like threads, hardware objects may have priorities, deadlines, and contexts. It is the job of the run-time environment to organize this information and make decisions based upon it.

Using software allows us to write our applications at a very high level of abstraction. For example, if the software needed to decompress an image, the attached FPGA could be reconfigured with the decompression algorithm and fed the data. To the main software application, this action is no different than asking an analog-to-digital converter to read a voltage and return the answer. The run-time environment software, however, is responsible for reprogramming the FPGA and executing the task.

Programming reconfigurable systems

Reconfigurable computing combines traditional software related topics as languages, compilers, operating systems, and libraries with hardware related topics of digital design.

Reconfigurable systems present a formidable challenge in terms of algorithm design tools. Design tools for FPGA devices, the building blocks of reconfigurable hardware, are oriented towards ASIC development environments, in which digital design engineers create large (multi-million gate), complex designs that, once created and validated, do not change. In contrast, reconfigurable supercomputers require a more software-centric development environment, in which algorithms are constantly revised and tested.

In response to the need for software-oriented tools, vendors and researchers have developed compilers for software programming languages that synthesize hardware. Compilers for several C variants, Java, and Matlab have become available in the past few years. The compiler must generate a structural hardware representation (such as VHDL-RTL) that represents the connections between units contained in a library, with direct correspondence to the operators of high-level programming languages.

Applications:

While commercial reconfigurable computing platforms are starting to become available, the majority of work has been done in a research context. There are some areas and problems that reconfigurable computing is ideal for, including:

- Real-time image analysis
- Pattern recognition
- Automatic target recognition
- Cryptography
- Computational biology
- Signal processing

One theoretical application is a smart cellular phone that supports multiple communication and data protocols, though just one at a time. When the phone passes from a geographic region that is served by one protocol into a region that is served by another, the hardware is automatically reconfigured. This is reconfigurable computing at its best, and using this approach it is possible to design systems that do more, cost less, and have shorter design and implementation cycles.

Heading into the future, evolvable hardware (EHW) is designed to adapt to changes in task requirements or changes in the environment, through its ability to reconfigure its own hardware structure dynamically and autonomously. This capacity for adaptation is achieved by employing efficient search algorithms known as genetic algorithms. Evolvable hardware has great potential for the development of innovative applications, including autonomous spacecraft and exploration systems.

Here's why space can be a "killer app" for Reconfigurable Computing:

- After launch, unmanned spacecraft electronics are generally unavailable for physical upgrade or repair. RC technology allows new hardware circuits to be uploaded via a radio link.
- New circuit configurations can overcome design faults, allow improved processing algorithms to be uploaded, or change system functionality in response to changing mission requirements. Combined with artificial intelligence applications, the unmanned spacecraft may be able to select circuits on its own to correct the problems.
- The same circuitry can be used with different configurations at different stages of a mission, reducing weight and power requirements.

- If part of an FPGA fails, then circuitry can be reprogrammed to make use of remaining functional portions of the chips.
- Use of FPGAs allows generic circuit boards to be designed, which are customized for individual applications. This helps overcome the very high NRE (non-recurring engineering) costs associated with small volume spacecraft design. Physical and environmental qualification costs can also be shared across many missions.
- In-flight reconfiguration provides additional safety margins for missions with very short lead-times, or for those where mission requirements are not fully defined at launch.

NASA Examples:

NASA Langley Research Center is one NASA installation exploring reconfigurable computing applications. They have developed a reconfigurable Field Programmable Gate Array (FPGA)-based research *hypercomputer* that is capable of performing comprehensive engineering and scientific calculations. Two modes have been adopted to exploit Langley's Star Bridge Systems HC-38 (and 2 HAL15s) for analysis calculations:

1. Rewrite legacy code for the hypercomputer to fully exploit parallelism
2. Use the hypercomputer to accelerate time-consuming (bottleneck) calculations

Software was entirely rewritten from C++ or Fortran, to take advantage of the parallelism inherent in the hypercomputer (approach 1). When only a small portion of a software application was computationally intensive, that portion was rewritten to the hypercomputer native language, and the rest of the code was left alone (approach 2).

The Jet Propulsion Laboratory (JPL) has an Evolvable Hardware Laboratory (<http://ehw.jpl.nasa.gov/>) which develops and demonstrates self-reconfigurable circuits that evolve directly in hardware on a VLSI chip. Self-reconfiguration endows devices with the flexibility of in-situ adaptation to unforeseen conditions and with enhanced fault-tolerance. The evolved circuits will perform complex signal processing functions, such as adaptive filtering and randomization. This development will break new ground and demonstrate a new paradigm for the design of adaptive computing (i.e., evolvable hardware). It will shape new tools and approaches, and it will also establish a technology base that others can use.

JPL also hosts the Center for Integrated Space Microsystems, which investigates multiple areas including system-on-chip and revolutionary computing technology. One subgroup is involved in reconfigurable computing, with the goal to evaluate new technologies that will increase spaceflight system computing capabilities and robustness through the use of in flight alterable hardware.

FedSat, an Australian science and engineering research satellite, was launched in 2002. One payload on FedSat is the Adaptive Instrument Module (AIM), which is a reconfigurable computer optimized for spacecraft instrument use. AIM has demonstrated autonomous instrument processing that is reconfigurable and adaptive. The use of the AIM enables reconfiguration of the FPGA circuitry while the spacecraft is in flight. This flexibility reduces mission risk, especially for missions with a very tight development schedule. The AIM is designed to either directly interface with sensors or instruments or to receive data through the spacecraft data handling system. AIM conducted a series of designed experiments, including a demonstration of implementing data compression, data filtering, and communication message processing and inter-experiment data computation.

The design of the AIM specifically addresses the concerns of using SRAM-based FPGAs in the space environment. The AIM demonstrates techniques to detect and remediate radiation-induced upsets in these FPGAs and will automatically restart in the event of an upset. The design has been proven in flight. When the module suffered a memory error due to the bombardment of cosmic radiation, AIM automatically detected and then reset itself. This prevented the memory error from causing an error in the data it was processing.

The team that developed AIM at the Applied Physics Laboratory/John Hopkins University is now working with NASA's Langley Research Center to take the next step in reconfigurable, self-repairing space borne computer design. The project is called ADAPT – Adaptive Data Analysis and Processing Technology. Because it's fully reconfigurable, an ADAPT computer can serve as the front-end package for virtually any type of instrument – for example, a spacecraft might carry six scientific instruments, each served by a physically identical, but differently programmed, ADAPT computer. As the design evolves, an ADAPT computer may carry up to 20 preprogrammed operating modes for controlling its instrument.

A prototype ADAPT computer has been built and is scheduled for testing aboard NASA's Proteus aircraft. Proteus holds records for peak altitude and sustained altitude in level flight and performs a variety of missions, including simulated space flight. It will fly an ADAPT computer that's incorporated into the Tropospheric Trace Species Sensing Fabry-Perot Interferometer developed under NASA Langley's Instrument Incubator Program.